Formal Verification of Key Exchange Protocol Security with Tamarin Prover

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Harrison J. Nicholls

May 2023

Approved for the Division
(Computer Science)

Chanathip Namprempre

Acknowledgements

To my family, for supporting me through my education, and enabling me to have the fortune of attending Reed. Thank you for giving me the best education possible.

To my advisor, Meaw Namprempre, for relentlessly encouraging me in the process of writing this. Thank you for opening my eyes to the world of academic research and believing that I could handle a topic so new to me.

To Sarah, for your support and companionship, and for sharing tea and snacks with me while we worked together late at night.

To my CS friends, for sticking together through the journey that was majoring in CS at Reed.

Table of Contents

List of Figures

Abstract

This thesis deals mostly with two topics: cryptographic key exchange and formal verification of cryptographic protocols. We begin with a thorough introduction to Tamarin Prover, covering the major components of a Tamarin model and providing a minimal example to build intuition for protocol specification with rewrite rules. We discuss specifics of writing model code and lemmas, and explore the process of automated theorem proving. We follow this with a discussion of the symmetric and asymmetric settings in cryptography, and the general problem of key exchange. Then, we present a specific, basic instance of a key exchange protocol, and we apply Tamarin to it. Finally, we explain and apply Tamarin to a simplified version of QUIC, a modern key exchange protocol developed by Google.

Introduction

Internet communication involves the sending and receiving of messages over unsecured networks. Many parties are able to view communications sent over public channels, and there is often a desire to encrypt these communications to protect them from being read by parties with malicious intent. Cryptography aims to study methods of communication in the presence of this adversarial behavior while making guarantees of security to the communicating parties. A cryptographic system is built of many small parts, each of which behaves and interacts with others in complex ways. Cryptographers often prove properties of these small parts, but when the parts are combined into a larger system it becomes increasingly difficult to guarantee security. A *protocol* in cryptography is a set of rules that specify how some group of parties is to communicate. Protocols rely on *schemes*, which in turn rely on cryptographic primitives like *block ciphers*. While proving security of the underlying schemes and functions in a protocol by hand may be manageable, determining whether the entire protocol is secure requires a great deal of time and effort: the interaction of multiple rules gives rise to significantly more complexity. Additionally, cryptographic schemes and primitives are often designed and reasoned about purely mathematically, often with insufficient consideration to their real-world implementation. On the other hand, protocols are often born first in implementation, and then people later seek to create proofs about their security. Because protocols are so inherently complex and tied to implementation details, manual cryptographic analysis is often error-prone, expensive, and limited by the level of abstraction used by the analyzer. In recent years, a technique known as *formal verification* has begun to be applied to cryptography. Formal verification, in essence, is a method of automating proofs with a computer. Its utility to cryptography is both in small scale projects where it is not feasible to involve a cryptographer, and in large-scale, real-world protocols that are complex enough for manual analysis to be impractical.

Tamarin Prover is a tool for cryptographic formal verification released in 2012 [1]. Its backend is written in Haskell. The primary mechanic for automatic proofs in Tamarin is a rule-based rewriting system, in which the language keeps track of a global state as a set of facts that can be rewritten according to the provided set of rules. For a user of Tamarin who seeks to analyze a protocol, there are three main elements to interact with: the model, the lemmas, and the proof. Tamarin is a model checker in its presentation to the user, but unlike other major protocol verification tools [2], its internal mechanism for generating proofs is a constraint solver.

A particularly interesting problem in cryptography (and one well-suited to formal verification for its relative ease of understanding) is key exchange. Many schemes with proven security properties rely on the existence of a shared secret key between multiple parties, and the establishment of this shared secret key is not a trivial task. A party of course cannot send the key over a public channel, but it also isn't possible to send an encrypted version of a key, as this would again rely on the prior existence of some shared secret key. In this paper, we present the 'Diffie-Hellman key exchange' [3], the first published method for securely exchanging cryptographic keys over an insecure communication channel. We then show a model in the Tamarin Prover language for Diffie-Hellman and explore some security results.

QUIC (Quick UDP Internet Connections) is a modern transport layer protocol developed by Google which has a key exchange phase [4][5]. In certain cases, QUIC enables a client and server who have communicated before to establish a new shared secret key while simultaneously sending data. This is known as 0-RTT (zero round-trip-time) connection resumption or handshake. Whereas a single round trip time would normally be spent on establishing a shared key before the transmission of data, QUIC eliminates this waiting time [5]. We present a simplified version of this mode of QUIC as a standalone protocol and apply Tamarin to it, examining liveness and security properties.

Chapter 1

Tamarin Prover

Tamarin aims to be on the completeness side of a theoretical completeness-vs-ease of use tradeoff, but it also provides an interactive means of viewing and working with proofs. Sometimes, the user needs to provide guidance through this interactive interface. Depending on the approach to modelling a protocol in Tamarin, different degrees of user assistance can be required.

1.1 Overview

While cryptographers are able to prove security of primitives such as encryption schemes through quantifiable proofs of properties like semantic security, protocols often give rise to significantly more complexity than can be reasoned about purely with symbolic manipulation on paper. Tamarin aims to offer an automated, symbolic solution to the problem of proving security properties of cryptographic protocols. Tamarin is a tool that allows a user to specify an entire protocol at some level of abstraction in terms of “rewrite rules.” These are rules that operate on a multiset of current facts in the system state. The user then writes logical lemmas that represent desired security properties, and uses an interactive interface to watch and guide Tamarin's engine in proving these lemmas.

1.2 Tamarin's Formal Verification

We briefly mention the formal verification methods used by Tamarin Prover. The problem of creating proofs for a set of logical statements (lemmas in Tamarin's case) which apply to a specified model is known as *model checking*. Internally, however, Tamarin Prover works as a *constraint solver*.

1.2.1 Constraint Solving

A constraint solver is a tool that solves *constraint satisfaction* problems. This is essentially the problem of finding a satisfying assignment of variables that have been constrained in terms of each other.

Letter Addition Problem

```
  S E N D
+ M O R E
---------
M O N E Y
```

This is a canonical example of a constraint satisfaction problem, sometimes called an 'alphametic problem.' The goal is simply to assign a number to each letter variable so that the addition is mathematically correct. In this case, the constraints on the values of each variable are defined by the given equation. The correct answer is the following assignment:

$$\begin{array}{rr} \mathsf{S}:9 & \mathsf{E}:5 \\ \mathsf{N}:6 & \mathsf{D}:7 \\ \mathsf{M}:1 & \mathsf{O}:0 \\ \mathsf{R}:8 & \mathsf{Y}:2 \end{array}$$

In solving constraint satisfaction problems, it is often much too costly to check permutations on the variables. In practice, constraint solvers often use heuristics and other optimizations to reduce the search space.

1.2.2 Model Checking

A model checker is a tool that checks whether a model of a system (in essence, a description of some finite state machine) meets a set of specifications. Typically, model checkers don't do an exhaustive search to prove properties, but rather represent the system using some formula in propositional logic and attempt to simplify it. There are many advanced techniques that simplify systems or bound the number of steps to make the problem easier to solve.

1.3 Components of the Model

Protocols are specified in Tamarin using a rule rewriting system of logical premises and conclusions. The logical premises include some facts about the current state of the parties in the network, and the conclusions then modify the state in some way. We can also associate rules with certain temporal variables (variables that refer to a point in time and can be compared to other temporal variables) using "action facts," which become useful in the lemmas section. For the proof generation, what Tamarin is doing in essence is taking the negation of the lemmas we want to be true, and then finding a violating "trace." A violating trace is an ordering of application of rewrite rules on the global state that results in the negation of the lemma being satisfied.

1.3.1 Facts

A fact is simply an object associated with variables of certain types. Syntactically, a fact looks like the following:

```
  FactName(~A,!B,C)
```

where the name `FactName` is the name of the fact, and the names inside refer to typed variables. We refer to the binding of the typed variables listed in the specified order with the name of the fact as a *signature*.

1.3.2 Rules

A rule consists of a premise, an optional action fact (not shown), and a conclusion. Syntactically, a rule looks like this:

```
rule Name:
  [leftside(A)]
  -->
  [rightside(A)]
```

The left side of the rule (shown here above the arrow) is the premise, and specifies the facts that must exist in the system's state and be *matched* for the rule to be applied. The right side of the rule is the conclusion. It specifies which facts will be written into the system's state when the rule is applied. When evaluating the left side of the rule, Tamarin does pattern-matching on any fact in the system state that might match the signature of a required rule. Specifically, a match is possible where two facts have the same name and are bound to typed variables listed in the same order. This way, variables in the same position in a fact can simply be renamed within the context of a rule. Generally, when Tamarin applies a rule, the facts on the left side are deleted from the state, and the facts on the right side are added to the state. The one exception to this is provided deliberately by persistent facts (see Section 1.6).

1.3.3 Action Facts

Like a normal fact, an action fact is an object associated with variables, but it also serves a vital role in proofs. Syntactically, action facts appear between the left and right hand side of rules in the following form:

```
  --[Fact(A,B,C)]->
```

The purpose of an action fact is to appear on the trace. Tamarin Prover's job is to explore different traces, searching for violations of properties. Specifically, if a rule with an associated action fact is applied at any time during a particular execution, it appears on the trace associated with that execution. This is useful because our lemmas make claims that refer directly to these action facts, requiring properties such as temporal ordering or inequality.

1.3.4 Fresh Facts

New variables can be introduced to the Tamarin state through the use of the "fresh" keyword. Syntactically, a rule that uses a fresh fact looks like the following:

```
rule introduce_n:
  [ Existing(x), Fr(~n) ]
  -->
  [ New(x,~n),Out(<x,~n>) ]
```

Even though the fresh variable `n` did not exist in the state before the application of `introduce_n`, we were able to introduce it through the use of the `Fr` keyword.

1.3.5 The Public Channel

Tamarin's modelling of protocols assumes one public channel, to which all parties in the system have access. By default, the adversary is a Dolev-Yao [6] adversary that can intercept and modify any message being sent over the public channel. Syntactically, we interact with Tamarin's public channel through the keywords `In()` and `Out()`. For example:

```
  rule Reveal_ltk:
  [ !Ltk($X, ltk) ]
```

```
    -->
  [ Out(ltk) ]
```

Rule `Reveal_ltk` represents the instance in which the long-term key associated with party `$X` is revealed to the adversary. Similarly, a rule's left hand side could contain `In()` with some signature, and this rule could be applied after a fact with `Out()` in its conclusion had been applied.

1.3.6 Lemmas

A lemma is a property written explicitly in boolean logic in terms of action facts, variables, and temporal variables. To analyze a protocol in Tamarin, we write security properties in the lemmas section. Lemmas are specified in reference to the temporal variables associated with certain rules in the protocol. For example, a lemma might state that for all session keys k, there exists a server that answered a request, and no other client had the same request, and so on. Proofs take into account the rules that specify the execution and the black box equations that are defined.

1.4 Records in the System

Tamarin maintains four internal structures through any execution of a proof: the *state*, the *trace*, the public channel, and the set of instantiations of variables known by the adversary. The last item refers specifically to the variables associated with a particular application of a rule and only to variables within action facts on the trace.

1.4.1 State

The state of the system simply represents an unordered multiset — also referred to in documentation as a "bag" — of facts. Consider the following rewrite rule:

```
rule Demonstrate_state:
  [Fr(x),Fr(y)]
    -->
  [Fact_1(x),Fact_2(y)]
```

At the end of a valid trace for a theory where `Demonstrate_state` is the only rule and is applied exactly twice, the state could look like:

{`Fact_1(X.1)`, `Fact_2(X.2)`, `Fact_2(Y.1)`, `Fact_2(Y.2)`}

Notice that the state can contain multiple facts of the same name, but they are distinguished by the instantiations of variables (as unique constants) bundled in the facts.

1.4.2 Trace

The trace represents a partial ordering of action facts corresponding to the application of rules. In a trace, some rules must have been applied in a certain order, and others do not have this requirement. The trace is the actual object that Tamarin prover interacts with when proving lemmas. More specifically, Tamarin searches through the space of possible traces given the rewriting rules, and formulates a proof. If the lemma is not satisfied, Tamarin simply needs to find a trace that violates the lemma. If the lemma is satisfied, Tamarin will disprove the *negation* of the lemma. The total set of possible traces is defined by the rules specifying the protocol.

1.4.3 Public Channel

The public channel (see Section 1.3.5) is a structure that tracks all messages sent during an execution of the protocol. Messages over the public channel are pattern-matched in the same way as rule signatures (see Section 1.3.2).

1.4.4 Known Variables

The set of known variables simply represents the instantiations of variables that the adversary is able to compute the value of at a certain time. In lemmas, we can check membership of this set with the syntax:

```
  K(x) @ #i
```

indicating that the adversary knows the variable `x` at time `i`. To express secrecy of a variable, one could specify:

```
  not (Ex #j. K(x)@j)
```

More precisely, `K` is an action fact that is implicitly defined for any variable in a model. Tamarin includes the following builtin rule to allow this:

```
rule isend:
  [!KU(x)]
  --[K(x)]->
  [In(x)]
```

The rule `isend` being applied means that the adversary is able to call `Out` on variable `x`, so that the conclusion of the rule is valid. The persistent fact `!KU(x)` denotes that the adversary knows `x`.

1.5 Minimal Example

1.5.1 First Attempt

To illustrate the role of the trace in Tamarin's proofs, we will examine a minimal example of a Tamarin model. This example is purely illustrative, and it does not contain any cryptographic components. The idea is to write three rules which can logically be applied in only one order by the nature of the facts they specify. The first rule creates a fact that is necessary for the second rule to be applied, which in turn creates a fact that is necessary for the third rule to be applied. We provide only three rules.

```
theory Minimal
begin

rule create_fact:
  []
  --[Create()]->
  [Fact(A)]
```

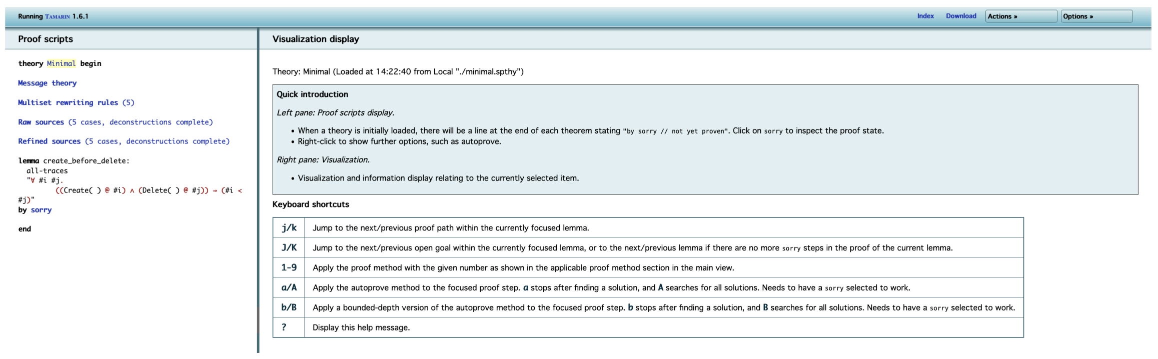
This rule allows Tamarin to add a fact of signature `Fact(x)` to the state. It is associated with action fact `Create()`, which has no associated variables.

```
rule consume_and_create:
  [Fact(B)]
  --[Consume()]->
  [New_Fact(C)]
```

This rule allows Tamarin to consume a fact of signature `Fact(x)` from the state, and add a fact of signature `New_Fact(y)`. It is associated with action fact `Consume()`, which has no associated variables.

```
rule delete:
  [New_Fact(D)]
  --[Delete()]->
  []
```

This rule allows Tamarin to consume a fact of signature `New_Fact(y)` from the state, replacing it with nothing. It is associated with action fact `Delete()`, which has no



Proof scripts

```
theory Minimal begin

Message theory

Multiset rewriting rules (5)

Raw sources (5 cases, deconstructions complete)

Refined sources (5 cases, deconstructions complete)

lemma create_before_delete:
  all-traces
  "∀ #i #j.
        ((Create( ) @ #i) ∧ (Delete( ) @ #j)) ⇒ (#i <
#j)"
by sorry

end
```

Visualization display

Theory: Minimal (Loaded at 14:22:40 from Local "./minimal.spthy")

Quick introduction

Left pane: Proof scripts display.

- When a theory is initially loaded, there will be a line at the end of each theorem stating `"by sorry // not yet proven"`. Click on `sorry` to inspect the proof state.
- Right-click to show further options, such as autoprove.

Right pane: Visualization.

- Visualization and information display relating to the currently selected item.

Keyboard shortcuts

j/k	Jump to the next/previous proof path within the currently focused lemma.
J/K	Jump to the next/previous open goal within the currently focused lemma, or to the next/previous lemma if there are no more `sorry` steps in the proof of the current lemma.
1-9	Apply the proof method with the given number as shown in the applicable proof method section in the main view.
a/A	Apply the autoprove method to the focused proof step. **a** stops after finding a solution, and **A** searches for all solutions. Needs to have a `sorry` selected to work.
b/B	Apply a bounded-depth version of the autoprove method to the focused proof step. **b** stops after finding a solution, and **B** searches for all solutions. Needs to have a `sorry` selected to work.
?	Display this help message.

Figure 1.1: Opening a Theory in Tamarin

associated variables.

```
lemma create_before_delete:
"All #i #j.
    Create() @ #i & Delete() @ #j
    ==>
    #i < #j
"
```

Lemma `create_before_delete` asks Tamarin to prove a common sense statement: that if rules `create` and `delete` are applied, `create` must be applied first. It seems that since for `delete` to be applied, there must be a `New_Fact` fact in the state first, we can trace this logic backwards to conclude that `create_fact` must be applied before `delete`.

Proof

Now, we run this model in Tamarin to check the lemma. This is also a good opportunity to explore some of the features of Tamarin's interactive proof interface. In a terminal, we run:

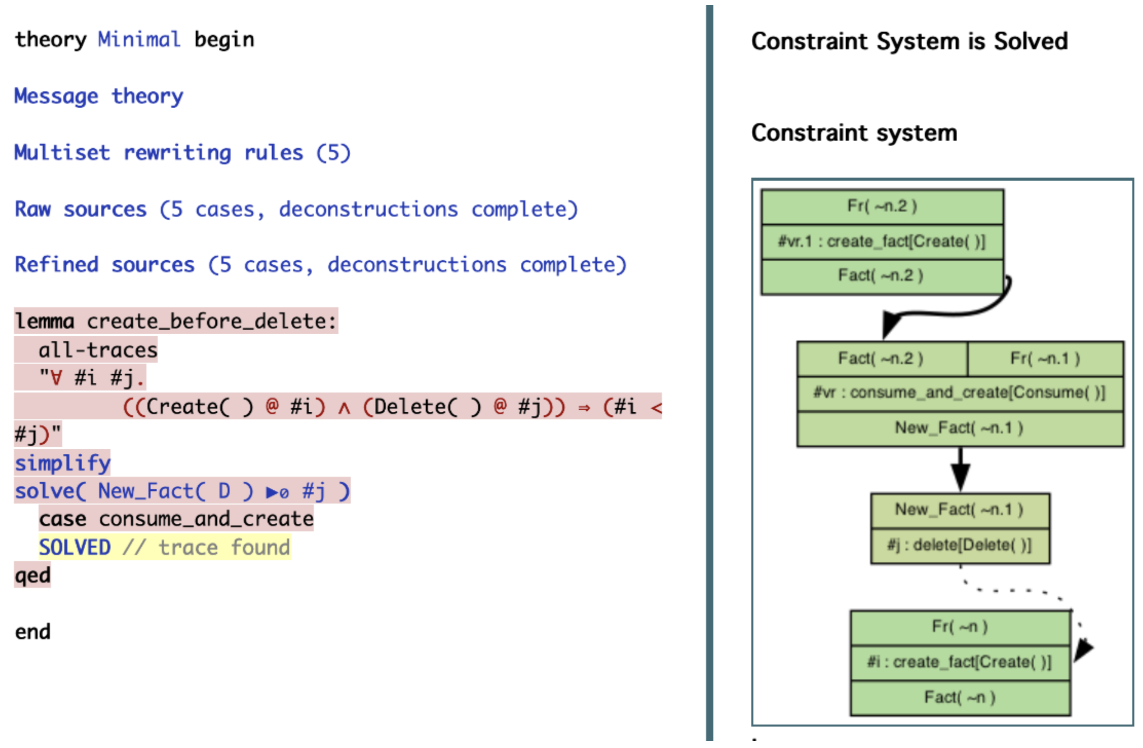
```
  > tamarinprover interactive minimal.spthy
```

And see the terminal message:

```
  The server is starting up on port 3001.
  Browse to http://127.0.0.1:3001 once the server is ready.
```

which starts a local server that presents a browser-based interface (see Figure 1.1).

Now, when we prompt Tamarin to prove the lemma, we see the result in Figure 1.2. It turns out that our intent for the model does not actually match its logic. Tamarin



```
theory Minimal begin

Message theory

Multiset rewriting rules (5)

Raw sources (5 cases, deconstructions complete)

Refined sources (5 cases, deconstructions complete)

lemma create_before_delete:
  all-traces
  "∀ #i #j.
        ((Create( ) @ #i) ∧ (Delete( ) @ #j)) ⇒ (#i <
#j)"
simplify
solve( New_Fact( D ) ▶₀ #j )
  case consume_and_create
  SOLVED // trace found
qed

end
```

Constraint System is Solved

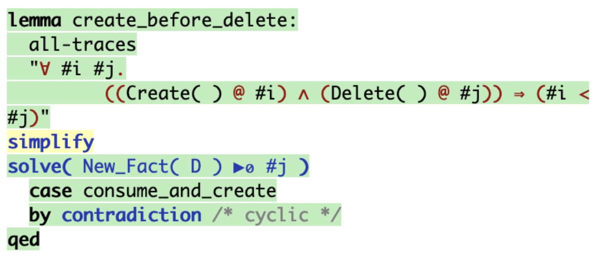
Constraint system

Figure 1.2: Proving a Lemma in the Minimal Theory

found a trace violating this lemma in which it applies the three rules in the expected order and then simply applies the rule `create_fact` for a second time, *after* `delete` has been applied. We can see from Figure 1.2 that it associates the second instance of action fact `Create` on the trace with time `#i` to show a precise instance of a violating trace for the lemma. Because the proof is of a violation of a lemma, Tamarin only needed to show one violating trace.

1.5.2 Fixing the Model

Tamarin provides a specific type of lemma called a *restriction* to address this problem. We could also modify `create_before_delete` to achieve the same effect, but in the case that we wanted to prove more than one lemma, we would need to modify every one of them. This would make the lemmas section more verbose and unnecessarily complicated. The purpose of a restriction is to limit the set of traces that Tamarin searches on when proving lemmas. A restriction simply states a property that a trace must satisfy to be considered in later proofs. In this minimal case, we are expecting the `create_fact` rule to be applied only once. To express this property in Tamarin's



```
lemma create_before_delete:
  all-traces
  "∀ #i #j.
         ((Create( ) @ #i) ∧ (Delete( ) @ #j)) ⇒ (#i <
#j)"
simplify
solve( New_Fact( D ) ▶₀ #j )
  case consume_and_create
  by contradiction /* cyclic */
qed
```

Figure 1.3: Proof of Lemma in Revised Minimal Theory with Restrictions

sorted first-order logic, we can use the following:

```
restriction create_once:
"All #i #j.
    Create() @ #i & Create() @ #j
    ==>
    #i = #j
"
```

The logical statement here is that for an arbitrary pair of applications of the `Create` rule associated with timepoints `#i` and `#j`, it must be the case that `#i` = `#j`. This means that any trace with more than one application of `Create` will be excluded from consideration, since they will of course have to be associated with distinct timepoints.

Running the theory and prompting Tamarin to prove lemma `create_once` again, we see the result in Figure 1.3. Note the lack of trace diagram: this is because Tamarin's claim is that there does not exist a violating trace to the lemma under the current restrictions, and that it is therefore true.

We can prompt Tamarin to expand on its explanation, by selecting "contradiction." We see the image shown in Figure 1.4, indicating the same logic we arrived at intuitively in designing the example. The solid arrows indicate necessary temporal ordering, and the dashed arrow indicates the temporal ordering that would be required to violate the lemma. We can see that it is impossible for `#i` > `#j`.

We can further probe Tamarin's interface to understand its reasoning for the proof. The *Refined Sources* section shows the preconditions for the rules in the model to be applied. Figure 1.5a shows that `create_fact` has no preconditions, and can simply be applied on an empty state. Figure 1.5b shows that `consume_and_create` has the precondition of a `Fact` fact existing on the state, which can only arise with `create_fact` having just been applied.

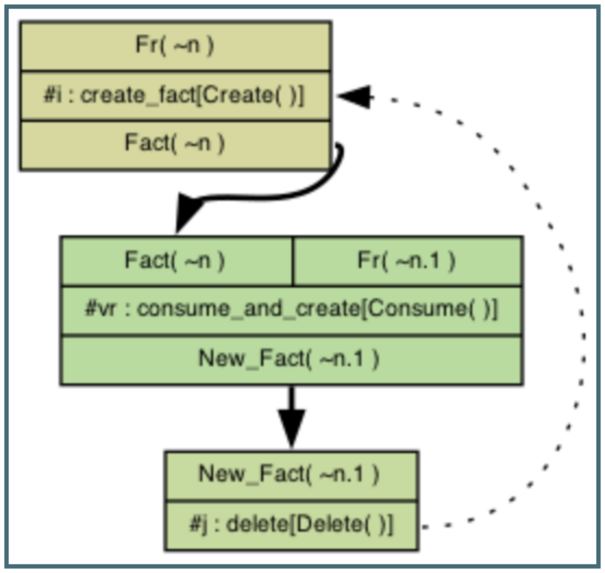


Figure 1.4: Explanation of Proof in Minimal Theory

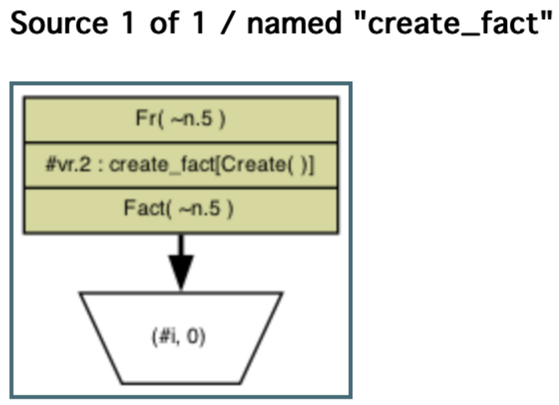
1.6 Sorts

Variables in Tamarin have a limited set of types. Tamarin refers to these types as "sorts" (a term from specific algebra domains), but they are types in the context of Tamarin modelling. A variable's sort must be consistent throughout its invocations, and casting between sorts is not possible.

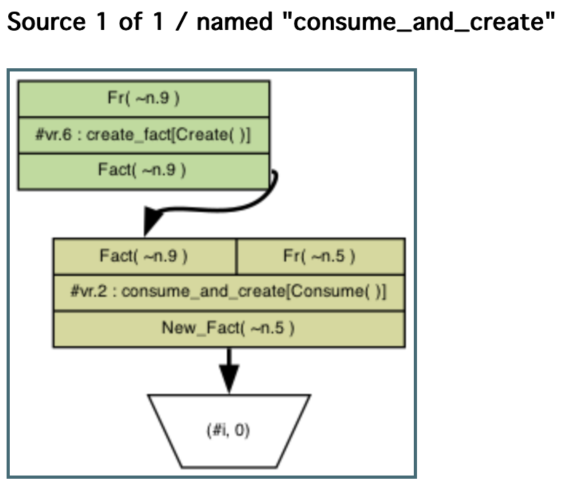
Note that from this point, font will be used as an element of notation to indicate whether a variable or name is being referred to symbolically in Tamarin, or computationally as in a real implementation of the protocol. We use `x` to represent a Tamarin name, and x to represent the true value of variable or function that will be computed if the protocol is executed by real parties.

1.6.1 Persistent

Syntax `!F` denotes that F is a persistent fact. While facts are normally removed from the state when a rule is applied if they appear in the left side of a rule and not the right, persistent facts always remain in the state.


Source 1 of 1 / named "create_fact"
Fr(~n.5)
#vr.2 : create_fact[Create( )]
Fact(~n.5)
(#i, 0)

(a) Raw Sources: `create_fact`


Source 1 of 1 / named "consume_and_create"
Fr(~n.9)
#vr.6 : create_fact[Create( )]
Fact(~n.9)
Fact(~n.9)
Fr(~n.5)
#vr.2 : consume_and_create[Consume( )]
New_Fact(~n.5)
(#i, 0)

(b) Raw Sources: `consume_and_create`

Figure 1.5: Raw Sources

1.6.2 Fresh

Syntax `~x` denotes that x is a fresh variable. Fresh variables are meant to model the properties of a random, distinct new number in a cryptographic protocol. The fresh type can be used for things like keys and nonces.

1.6.3 Public Name

Syntax `$S` denotes that S is a public name. A public name is known by all parties in a protocol, such as the name of a server or a client.

1.6.4 Message

Syntax `m` denotes that m is simply a ‘message’. Tamarin documentation is somewhat ambiguous in addressing why this sort is referred to as a message. The message sort can represent the objects being passed between parties, but it is also a sort of default type with no additional properties.

1.6.5 Public Constant

Syntax `‘c’` denotes that c is a public constant. They are known by all parties in the protocol, including adversaries. For example, a Diffie-Hellman group generator g would be syntactically represented as Syntax `‘g’`

1.6.6 Temporal Variable

Syntax '`#i`' denotes that `i` is a temporal variable. The temporal sort only applies to certain variables within lemmas (see Section 1.5 for example). It is a variable that refers to a point in time (in a trace) relative to other points in time. Temporal variables have binary comparisons `=`, `<`, and `>` implemented as operations on their type.

1.7 Adversary Type and Assumptions

Tamarin models the adversary as a Dolev-Yao adversary (Dolev & Yao 1983) by default. Every model works under the assumption that all parties, including the adversary, have access to a common broadcast channel, so everything passed to a call of `Out` is visible to an adversary.

Chapter 2

Two-party Key Exchange Protocols

In this chapter, we discuss how key exchange involves symmetric and asymmetric-key cryptography. We also introduce some underlying concepts, particularly in the symmetric setting.

2.1 Asymmetric and Symmetric Settings

In this section, we present the two settings in cryptography in order to provide background for a type of protocol that takes a communication from the first setting to the second.

2.1.1 Symmetric Setting

Symmetric key cryptography describes the setting in which trusted parties share a secret key k prior to communicating. This key k is used by all trusted parties both to encrypt[1] messages with some encryption function and also to decrypt them. We will introduce AEAD (*authenticated encryption* with *associated data.*) We can formalize the notion of symmetric scheme AEAD as follows [7]:

Let AEAD be a symmetric encryption scheme. AEAD contains a triple of algorithms KG, Enc, and Dec. It also has the associated sets $\mathsf{Message} \in \{0,1\}^*$, $\mathsf{Nonce} \in \{0,1\}^n$ for a fixed n, $\mathsf{Header} \in \{0,1\}^*$, and $\mathcal{K}$: the key space.

[1]Encryption is not the only goal of schemes within the symmetric setting, or in cryptography in general: there are also schemes designed for other purposes such as authentication of messages. We will only talk about a form of encryption to give an example of syntax and definitions.

KG: Key Generation

KG is a randomized algorithm that takes no inputs and returns a secret key $k \in \mathcal{K}$.

Enc: Encryption

Encryption algorithm Enc takes a key $k \in \mathcal{K}$, $N \in \mathsf{Nonce}$, $H \in \mathsf{Header}$, and plaintext $m \in \mathsf{Message}$. It returns a ciphertext $c = \mathsf{Enc}_K^{N,H}(m)$ or a special symbol $\perp$ indicating failure as output.

Dec: Decryption

Decryption algorithm Dec takes k, $N \in \mathsf{Nonce}$, $H \in \mathsf{Header}$, and ciphertext c and returns plaintext $\mathsf{Dec}_k^{N,H}$, which is either a string in $\mathsf{Message}$ or $\perp$, as output.

Correctness

It is required that

$$\forall k \in \mathcal{K}, N \in \mathsf{Nonce}, H \in \mathsf{Header}, c \in \{0,1\}^* : \mathsf{Dec}_k^{N,H}(\mathsf{Enc}_k^{N,H}(m)) = m.$$

Security

We generally use two security definitions for AEAD: PRIV and AUTH. Informally, $\mathsf{Adv}_\Pi^{\mathsf{PRIV}}(A)$ [1] is measured as the scaled and shifted probability (scaled and shifted so that a 0 value for advantage corresponds to the advantage of an adversary whose guesses are no better than random) that adversary A is able to distinguish between a random bitstring and a ciphertext generated by Enc_k for some key k.

Advantage $\mathsf{Adv}_\Pi^{\mathsf{AUTH}}(A)$ deals with an adversary A with access to encryption oracle Enc_k for some key k who tries to output $(\mathsf{N}, \mathsf{H}, \mathsf{C})$ where $\mathsf{Dec}_k^{N,H} \neq \perp$. This is called a forgery. We say that AEAD scheme Π is secure if $\mathsf{Adv}_\Pi^{\mathsf{PRIV}}(A)$ and $\mathsf{Adv}_\Pi^{\mathsf{AUTH}}(A)$ are small for a reasonable adversary A.

2.1.2 Asymmetric Setting

Asymmetric key cryptography, also known as *public-key cryptography*, is the setting in which each party has a pair of mathematically related keys: a private key and a public key. Unlike in symmetric key cryptography, where the same key is used for

[1] The *advantage* of adversary A against AEAD scheme Π under security definition PRIV

both encryption and decryption, asymmetric key cryptography uses these two distinct keys.

For each party, the public key is visible to all other parties (including adversaries) and can be used by anyone to encrypt data that is intended to be sent to the owner of the corresponding private key. The private key is kept secret by the owner and is used for decrypting the data that has been encrypted with the public key. An important requirement of public-key cryptography is that the public key cannot be used to determine the private key or to decrypt messages.

Generally, public key encryption (PKE) schemes which are secure under chosen ciphertext attacks are considered strong schemes in the asymmetric setting. See [8] for a formal presentation of syntax, correctness, and security of PKE. We will not formalize these concepts in this paper.

2.2 Key Exchange

Key exchange describes the process by which two parties securely exchange cryptographic keys. The problem key exchange aims to solve is creating an environment in which symmetric cryptography is possible. Symmetric key cryptography requires a shared secret key, but aside from making two parties agree on a key outside of the network (e.g. in person), the most commonly used way to establish a shared key is with asymmetric cryptography. In Section 3.1, we present a standard asymmetric setting approach to key exchange.

Chapter 3

Tamarin for Two-Party Key Exchange

In this chapter, we first present the Diffie-Hellman key exchange protocol. We then examine the process of modelling it in Tamarin Prover, discussing the rewrite rules and how they correspond to a real-world execution of the protocol.

3.1 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange protocol is a classic key exchange protocol (see Section 2.2) which can illustrate the complexity of designing protocols in general.

Alice and Bob want to establish a shared secret key that they can use to encrypt messages and send to one another with a symmetric scheme like AEAD (see Section 2.1.1). The idea is for Alice and Bob, after exchanging a few messages over a public, unsecured channel, to end up with this shared secret. We start with two values that are known by everyone: p, a large prime number, and g, which is a generator in a cyclic group. In this case we use the cyclic group $\mathbb{Z}_p^*$, which is the set of integers $\{1, ..., p-1\}$ and the operation multiplication modulo p. Then, Alice privately decides on a secret key a, chosen from this set, and Bob decides on his own secret key by the same method. At this point, Alice and Bob only know their own secret keys.

Now, Alice computes a new value, which is the public generator composed with itself by the group operation a times, modulo p ($g^a \bmod p$), and then sends this value over the network. The important detail here is that this does not leak Alice's private key to the adversary, because computing a from A is an instance of the discrete log problem, which is hard to compute on some large cyclic groups. Bob does the same,

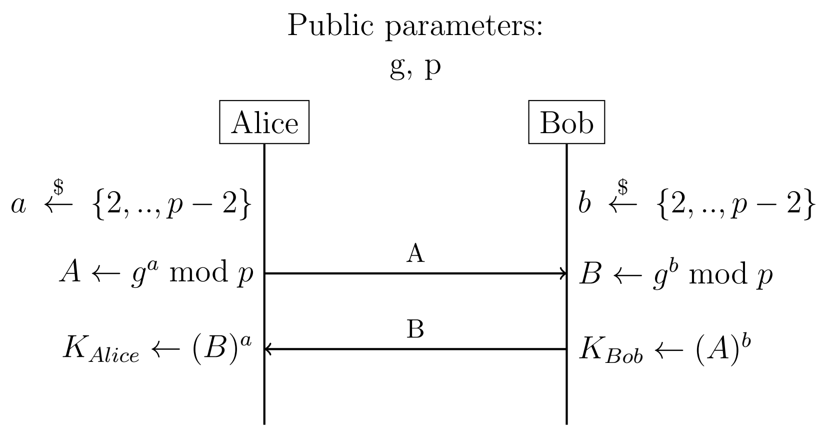

Public parameters:
g, p
Alice
Bob
$a \overset{\$}{\leftarrow} \{2,..,p-2\}$
$b \overset{\$}{\leftarrow} \{2,..,p-2\}$
$A \leftarrow g^a \bmod p$
A
$B \leftarrow g^b \bmod p$
$K_{Alice} \leftarrow (B)^a$
B
$K_{Bob} \leftarrow (A)^b$

Figure 3.1: Diffie-Hellman Key Exchange. Notice that $K_{Alice} = K_{Bob} = g^{ab}$ and we are operating in the fixed group $\mathbb{Z}_p^*$. We write $x \overset{\$}{\leftarrow} X$ to denote that x is chosen uniformly from set X.

and now both Alice and Bob know their own secret keys and the value they just received from the other party. For the last step, they raise the received value to their secret key, and since we are working within a group, the values they calculate are equal by associativity.

$$\begin{aligned} K_{Alice} &= B^a \\ &= (g^b)^a \\ &= g^{ba} \\ &= g^{ab} \\ &= (g^a)^b \\ &= (A)^b \\ &= K_{Bob} \end{aligned}$$

Although this works under certain conditions, there is a vulnerability. A man-in-the middle attack can easily compromise the entire protocol. Specifically, an adversary *Eve* under the Dolev-Yao model can intercept every message and synthesize a replacement message to pass on to the other party [6]. For example, consider replacing party Bob in Figure 3.1 with adversary Eve. Alice has no way of verifying that the party she is communicating with is Bob, and Eve can simply carry out the key exchange choosing their own value for b.

3.2 Rules

We further introduce protocol modelling using Tamarin rewrite rules by modelling the Diffie-Hellman key exchange protocol described in the previous section.

3.2.1 Preamble

At the very beginning of the model, we start with some boilerplate code.

```
1 theory BasicDiffieHellman
2 begin
3
4 builtins: diffie-hellman
```

This starting code indicates that the name of the theory we are modelling is `BasicDiffieHellman`, and the syntactic element `begin` simply indicates the beginning of the model. Importantly, Tamarin includes a builtin called `diffie-hellman` that allows for explicit modelling of Diffie-Hellman exponentiation (see Section 3.1) with the `^` operator. Line 4 specifies this as an included builtin to indicate that instances of this operator should be parsed as such.

3.2.2 Create_Identities

Our first rewriting rule takes an empty system state and creates the identities of the parties that will act in the protocol.

```
rule Create_Identities:
  let
    pk = ‘g’^~sk
  in
  [Fr(~sk)]
    -->
  [Out(pk),!Identity($A,~sk,pk)]
```

The only fact on the left side of the rule is `Fr(~sk)`, which does not require anything to be in the current state. (see Section 1.3.4). Since it is a fresh fact, it instantiates the variable `~sk`. We are relying on the fact that both parties in this simple case of Diffie-Hellman key exchange simply have a public/private key pair and no other associated information. This way, we only need one rule to add both necessary identities—the client and the server—to the state. According to the definition of Diffie-Hellman, and as is understood in public-key cryptography, the public key of any party is required to be some function of its secret key.

In lines 1-3, we use Tamarin's `let` construct to specify this constraint. Another feature provided by Tamarin's `diffie-hellman` builtin is the syntactic notion of using `‘g’ ^A`, where g is the public constant being used for exponentiation, and $A \in \mathbb{Z}_p^*$. Recall that `‘c’` denotes that `c` is a public constant. (see Section 1.3.4)

The conclusion of this rule has two effects: the addition of persistent fact `!Identity($A,~sk,pk)` to the state and the broadcasting of `pk` on the public channel. The persistent identity is associated with `$A`, some public name. It contains a secret key and public key associated with the named party. We use `Out(pk)` to model the fact that every party on the network has access to any party's public key. Both the intended party and the adversary can access the public key from the common channel.

3.2.3 Client_Hello

`Client_hello` models the client's first message to the server.

```
rule Client_Hello:
  [!Identity($C,~C_sk,C_pk),!Identity($S,~S_sk,S_pk)]
    -->
  [Out(<‘client_hello’,$C,$S,C_pk>)]
```

It corresponds with the first flow in Figure 3.1. In practice, this message simply contains the client's public key, calculated as `‘g’` raised to its private key. We will see that Tamarin's representation needs to be slightly different.

First, the premise of this rule contains two facts. They are both `!Identity` facts, which is an attainable scenario if Tamarin executes `Create_Identities` at least twice. In the rule's premise, we assign different variables within the `!Identity` facts, but Tamarin pattern-matches these with any existing `!Identity` fact in the multiset. This is why we are able to start with a rule that produces an identity with associated public name `$A`, and still use names `$S` and `$C`.

Having required the existence of these two parties, we send a client hello on the public channel, which (most importantly) contains `C_pk`: the client's public key. We represent the message as a tuple within angle brackets, and include the client and server public names for convenience when distinguishing between messages.

Another detail to note is that since the `!Identity` facts are persistent, we do not need to include them on the right side of the rule, and they will remain in the fact multiset (see Section 1.4).

3.2.4 Server_Response_And_Get_Session_Key

The next step in the protocol is for the server to respond to the client's initial message with its own public key. However, we also have to handle the server's internal state and representation of the information it has.

```
1 rule Server_Response_And_Get_Session_Key:
2   let
3     S_k = C_pk^~S_sk
4   in
5   [!Identity($S,~S_sk,S_pk),In(<‘client_hello’,$C,$S,C_pk>)]
6     --[ServerSession($S,$C,S_k)]->
7   [Session($S,S_k),Out(<‘server_hello’,$S,$C,S_pk>)]
```

Rule `Server_Response_And_Get_Session_Key` corresponds with the second flow in Figure 3.1. As expected, we have two facts in the premise: the server's identity and the expected receipt over the public channel of a client hello message. When the server receives this public key from the client, we model it as immediately creating a session key. In the `let` binding in lines 2-4, we calculate a session key as `C_pk ^~ S_sk`, which is equivalent to $(A)^b$ in Section 3.1.

In the conclusion of this rule, we store this calculated value in the fact called `Session($S,S_k)`, which gets associated with the same public name variable as the server. Then, we are finally able to send the server's response on the public channel using `Out()`.

The other notable feature of this rule is the action fact `ServerSession`, which we will use later to check properties of the protocol.

3.2.5 Client_Create_Session_Key

Rule `Client_Create_Session_Key` is the last step of the simple Diffie-Hellman protocol.

```
1 rule Client_Create_Session_Key:
2     let
3       C_k = S_pk^~C_sk
4     in
5     [!Identity($C,~C_sk,C_pk),In(<‘server_hello’,$S,$C,S_pk>)]
6     --[ClientSession($C,$S,C_k)]->
7     [Session($C,C_k)]
```

We require a client identity to be in the state, and listen on the public channel (line 5) for the `server_hello` message. We again use a `let` binding to obtain `C_k`, the intended session key. We create another action fact within this rule, also for property checking.

3.3 Lemmas

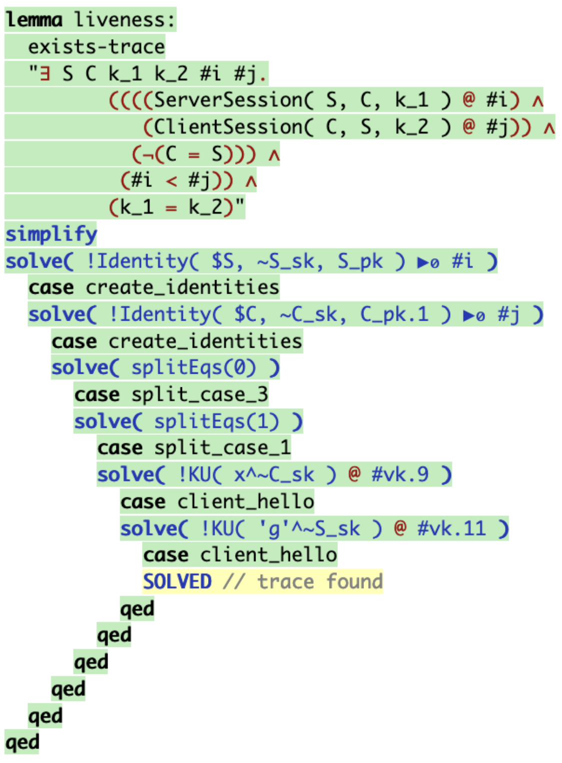
Now, we will examine two simple lemmas that can be applied to check properties of this model.

3.3.1 liveness

The `liveness` lemma asks Tamarin to show that it is possible for the protocol to be run in a way that results in an agreement on session keys between the two parties.

```
 1 lemma liveness:
 2 exists-trace
 3 "
 4   (
 5   Ex S C k_1 k_2 #i #j.
 6   ServerSession(S,C,k_1) @ #i &
 7   ClientSession(C,S,k_2) @ #j &
 8   not(C = S) &
 9   #i < #j &
10   (k_1 = k_2)
11   )
12 "
```

The `exists-trace` keyword indicates that Tamarin needs only to find one trace that satisfies its logical formula. Variables `k_1` and `k_2` (appearing after the existential quantifier in line 5 along with `#i` and `#j`) indicate each party's apparent session key at the time their message-sending rule is being applied. Temporal variables `#i,#j` serve to represent the order of rules being applied. By including `#i < #j`, we require that the `ServerSession` action fact appears on the trace before `ClientSession` does, which we expect in a normal execution of the protocol. Including `not(C = S)` prevents the case where the two parties have the same identity. Prompting Tamarin to prove the lemma, we see Figure 3.2, indicating that the lemma is satisfied by the model.



```
lemma liveness:
  exists-trace
  "∃ S C k_1 k_2 #i #j.
           ((((ServerSession( S, C, k_1 ) @ #i) ∧
              (ClientSession( C, S, k_2 ) @ #j)) ∧
             (¬(C = S))) ∧
            (#i < #j)) ∧
           (k_1 = k_2)"
simplify
solve( !Identity( $S, ~S_sk, S_pk ) ▶₀ #i )
  case create_identities
  solve( !Identity( $C, ~C_sk, C_pk.1 ) ▶₀ #j )
    case create_identities
    solve( splitEqs(0) )
      case split_case_3
      solve( splitEqs(1) )
        case split_case_1
        solve( !KU( x^~C_sk ) @ #vk.9 )
          case client_hello
          solve( !KU( 'g'^~S_sk ) @ #vk.11 )
            case client_hello
            SOLVED // trace found
          qed
        qed
      qed
    qed
  qed
qed
```

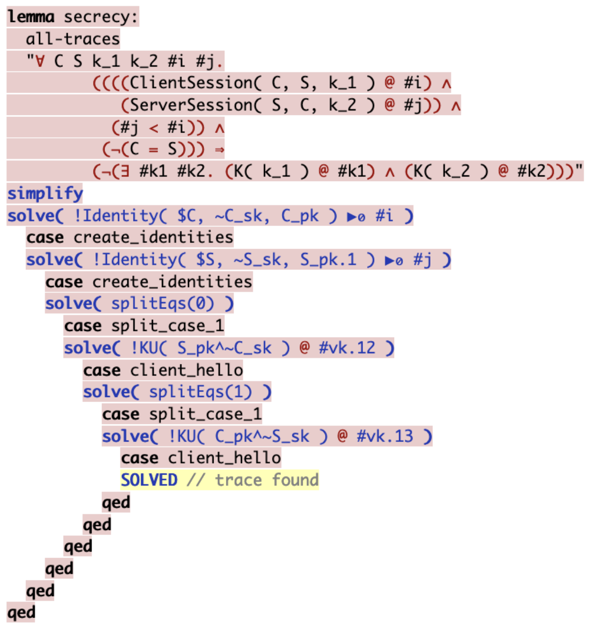
Figure 3.2: Tamarin result for `liveness`

3.3.2 secrecy

The `secrecy` lemma aims to check if for all traces (note the absence of `exists-trace`. Lemmas by default are checked in the `all-traces` case), the adversary is not able to discover the session key of either the client or server.

```
lemma secrecy:
  "
  All C S k_1 k_2 #i #j.
  (
    ClientSession(C,S,k_1) @ #i &
    ServerSession(S,C,fk_2) @ #j &
    #j < #i &
    not(C = S)
  )
  ==> not(Ex #l #m . K(k_1) @ #l & K(k_2) @ #m)
"
```

As described in Section 1.4, the `K(x)` keyword is another Tamarin namespace reserved action fact name which denotes that a theoretical adversary “knows” the variable `x`.



```
lemma secrecy:
  all-traces
  "∀ C S k_1 k_2 #i #j.
         ((((ClientSession( C, S, k_1 ) @ #i) ∧
            (ServerSession( S, C, k_2 ) @ #j)) ∧
           (#j < #i)) ∧
          (¬(C = S))) ⇒
         (¬(∃ #k1 #k2. (K( k_1 ) @ #k1) ∧ (K( k_2 ) @ #k2)))"
simplify
solve( !Identity( $C, ~C_sk, C_pk ) ▶₀ #i )
  case create_identities
  solve( !Identity( $S, ~S_sk, S_pk.1 ) ▶₀ #j )
    case create_identities
    solve( splitEqs(0) )
      case split_case_1
      solve( !KU( S_pk^~C_sk ) @ #vk.12 )
        case client_hello
        solve( splitEqs(1) )
          case split_case_1
          solve( !KU( C_pk^~S_sk ) @ #vk.13 )
            case client_hello
            SOLVED // trace found
          qed
        qed
      qed
    qed
  qed
qed
```

Figure 3.3: Tamarin result for `secrecy`

The implication that this lemma specifies with `==>` is that there do not exist two points in time such that the adversary knows the first key and the second key at the second time. The property that implies this simply includes the two relevant action facts which reference the two parties and their keys, using the builtin `K` to denote the adversary knowing each of the keys. As wee see in Figure 3.3, Tamarin is able to find a violating trace, and the lemma is not true. This is the result we expected from Section 3.1.

Chapter 4

Applying Tamarin to a Real World Protocol

In this chapter, we discuss QUIC, a protocol created by Google [5]. We present a simplified version of the protocol based loosely on material in [4], and then model it in Tamarin Prover as in the previous chapter.

4.1 The Protocol

QUIC (Quick UDP Internet Connections) is a transport layer protocol designed to be fast and extensible. QUIC is built on top of UDP (User Datagram Protocol), which allows for extensibility, as compared to TCP which is often implemented in the kernel. One of its most novel and interesting features is the 0-RTT (zero round-trip-time) connection establishment. At a high level, the idea of this mode of connection establishment is that the client remembers some configuration information from a previous session with the server that can be used to immediately begin securely transmitting data. To illustrate QUIC's 0-RTT operation, we consider SQUIC 0-RTT, a simplified QUIC-like protocol that abstracts away implementation details as shown in Figure 4.1.

The assumption before this protocol runs is that the client and server have established a connection prior to this execution, within some recency constraint. The client knows pk_s (the server's public key), which is equal to g^x, where x is the server's secret key.

The client then generates the following fresh values: a secret key y, and a nonce r_c. It computes $Y = g^y$. It also decides on a `cid`, which is a connection identifier. It then computes the Diffie-Hellman secret D_{C1} as $X^y = g^{xy}$. This will later be computed

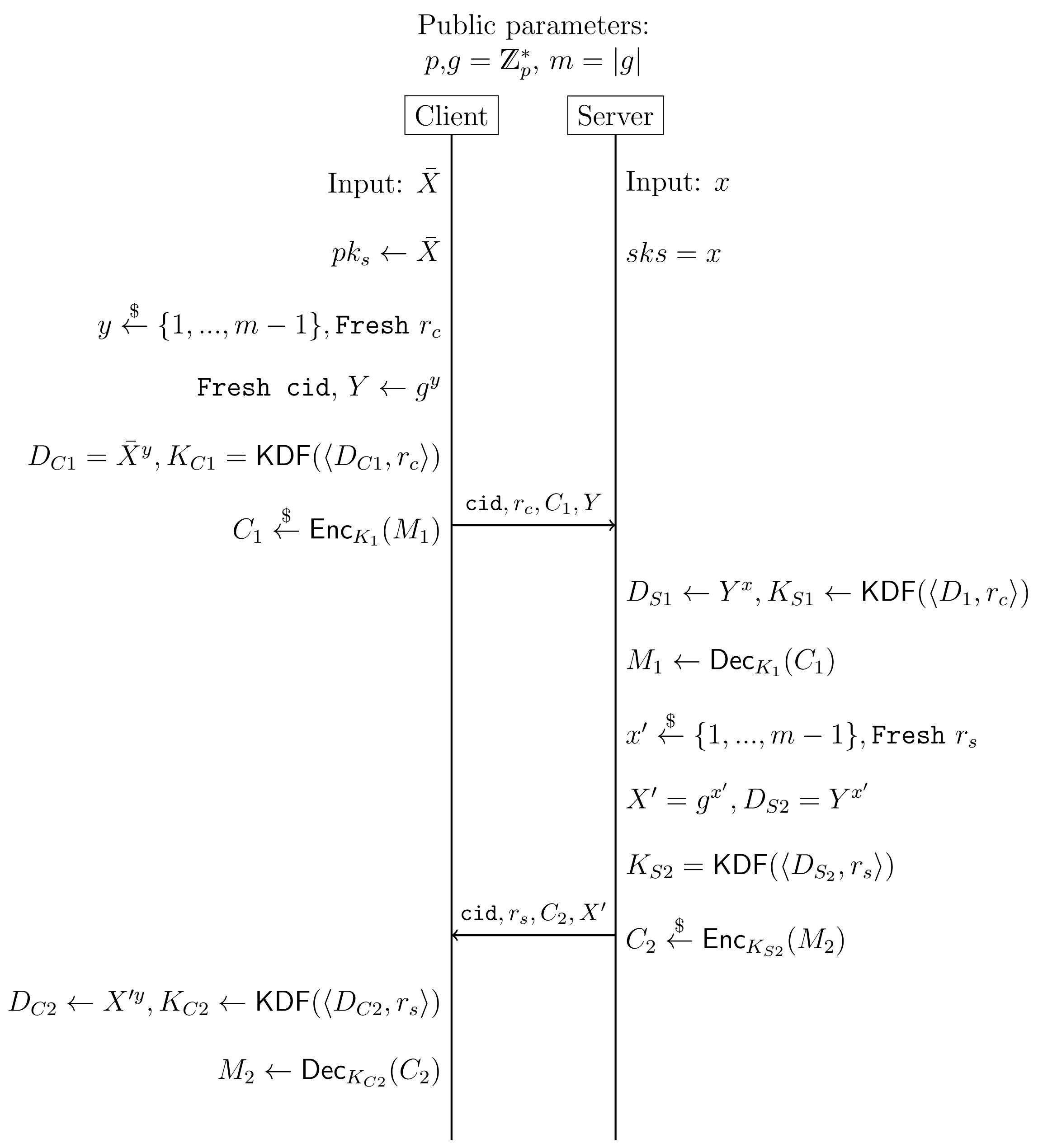

Public parameters:
$p, g = \mathbb{Z}_p^*$, $m = |g|$
Client
Server
Input: $\bar{X}$
Input: x
$pk_s \leftarrow \bar{X}$
$sks = x$
$y \xleftarrow{\$} \{1, ..., m-1\}$, $\texttt{Fresh}\ r_c$
$\texttt{Fresh cid}$, $Y \leftarrow g^y$
$D_{C1} = \bar{X}^y, K_{C1} = \mathsf{KDF}(\langle D_{C1}, r_c \rangle)$
$\texttt{cid}, r_c, C_1, Y$
$C_1 \xleftarrow{\$} \mathsf{Enc}_{K_1}(M_1)$
$D_{S1} \leftarrow Y^x, K_{S1} \leftarrow \mathsf{KDF}(\langle D_1, r_c \rangle)$
$M_1 \leftarrow \mathsf{Dec}_{K_1}(C_1)$
$x' \xleftarrow{\$} \{1, ..., m-1\}$, $\texttt{Fresh}\ r_s$
$X' = g^{x'}, D_{S2} = Y^{x'}$
$K_{S2} = \mathsf{KDF}(\langle D_{S_2}, r_s \rangle)$
$\texttt{cid}, r_s, C_2, X'$
$C_2 \xleftarrow{\$} \mathsf{Enc}_{K_{S2}}(M_2)$
$D_{C2} \leftarrow X'^y, K_{C2} \leftarrow \mathsf{KDF}(\langle D_{C2}, r_s \rangle)$
$M_2 \leftarrow \mathsf{Dec}_{K_{C2}}(C_2)$

Figure 4.1: SQUIC 0-RTT connection resumption protocol. Here, KDF is a key derivation function. Note that $\bar{X} = g^x$. $\texttt{Fresh}\ x$ denotes that x is being chosen uniformly from some domain. In reality, this domain is given by the technical specification of the protocol, and we expect secret values to be sufficiently long so as to be secure. In this model the values are assumed to be hard to guess, but not represented in a manner specific to their length.

by the server in much the same way as Diffie-Hellman, before being passed to key derivation function KDF.

In its final step before creating the ciphertext to be sent, it derives a session key using D_{C1} and the nonce r_c. In the real implementation of QUIC, an HMAC-based key derivation function (HKDF) with hash function SHA-256 is used. We will treat key derivation as a black box function, so the specifics of HKDF are not important to the Tamarin model. The input to KDF is represented in angled brackets to denote that the two values are being wrapped somehow (e.g. concatenated) into a single bitstring, as the arity of KDF is always 1.

After this setup, the client is ready to send its "client hello". It needs to send Y in order to achieve a shared session key with the server, but it also manages to send data over. It encrypts M_1 with its derived session key, yielding ciphertext C_1, and also sends r_1, the nonce used for this derivation. The other data it sends is `cid`, which allows both parties to track and associate messages with sessions.

Receiving these four objects over the public channel, the server now seeks to decrypt the first message, derive a new session key, and send some data of its own back to the client. In order to do this, it first needs to derive the ephemeral key K_1 that was just used by the client. Using group associativity for agreement on D_1, it derives K_{S1} by computing $\mathsf{KDF}(\langle Y^x, r_c\rangle)$. After recovering M_1 using this key and the agreed-upon encryption scheme, the server can begin deriving the new key K_2. As the client did in the first step, it computes $K_{S2} = \mathsf{KDF}(\langle Y^{x'}, r_s\rangle)$ for some fresh (random) values of x' and r_s. It sends the same relevant information over the public channel back to the client, completing the two flows in Figure 4.1. We have

$$\begin{aligned} K_{C2} &= \mathsf{KDF}(\langle D_{C2}, r_s\rangle) = \mathsf{KDF}(\langle X'^y, r_s\rangle) = \mathsf{KDF}(\langle g^{x'y}, r_s\rangle) \\ &= \mathsf{KDF}(\langle g^{yx'}, r_s\rangle) = \mathsf{KDF}(\langle Y^{x'}, r_s\rangle) = \mathsf{KDF}(\langle D_{S2}, r_s\rangle) = K_{S2}, \end{aligned}$$

implying that the server and client now share a session key.

4.2 Rules

We now describe the rewrite rules used in modelling SQUIC.

4.2.1 Preamble

We begin with similar boilerplate code to Section 3.1, but with two additions.

```
builtins: asymmetric-encryption, diffie-hellman
functions: kdf/1
```

The most important element here is `functions: kdf/1`, which defines a black box function of arity 1 called `kdf`. Although the math behind key derivation isn't stated in the model, Tamarin models it as a *perfect* cryptographic operation, meaning an adversary cannot learn anything about its input [9, Sec. 2].

4.2.2 Create_Server

Unlike in Diffie-Hellman, we do not write a single rule to create both identities.

```
  rule Create_Server:
    [ Fr(~x) ]
    --[ CreateServer() ]->
    [ Server($S,~x) ]
```

The starting state of each party (client and server) requires different knowledge and will therefore require different premises.

4.2.3 Create_Client

The `Create_Client` rule creates an identity for the client.

```
rule Create_Client:
    let
        X = 'g'^~x
        Y = 'g'^~y
        D1 = X^~y
        K1c = kdf(<D1,~rc>)
    in
    [ Fr(~y),Fr(~cid),Fr(~rc),Server($S,~x) ]
    --[ CreateClient() ]->
    [ Server($S,~x),Client($C,~y,Y,X,K1c,~rc,~cid), Out(Y),Out(X) ]
```

The client starts by knowing X, and it computes a new Y (see Figure 4.1). We use a let binding to ensure all of these relationships hold when the rule is applied. This rule contains everything before the first flow in Figure 4.1. Although this rule has a specific association with the client, we also take the server as a premise of the rule. We need to do this to relate the server and client's keys, and we also take this opportunity to call `Out` on each party's public key.

4.2.4 Client_Hello

Rule `Client_Hello` models the first flow in Figure 4.1.

```
rule Client_Hello:
    let
        c1 = aenc(K1c,<‘message’>)
    in
    [ Client($C,~y,Y,X,K1c,~rc,~cid) ]
    --[ ClientHello() ]->
    [ Out(<~cid,~rc,c1,Y >) ]
```

The let binding requires that C_1 be the encryption of some plaintext. The rule simply calls the `Out` builtin on the `c1` along with additional parameters `Y`, `~rc`, and `~cid`.

4.2.5 Server_Respond

Rule `Server_Respond` contains all of the logic between the first and second flow of Figure 4.1, and the conclusion of the rule results in the second flow.

```
rule Server_Respond:
    let
        D1 = Y^~x
        K1s = kdf(<D1,~rc>)
        Xprime = 'g'^~xprime
        D2 = Y^~xprime
        K2s = kdf(<D2,~rs>)
        c2 = aenc(K2s,<'message2'>)
    in
    [ In(<~cid,~rc,c1,Y>),Fr(~xprime),Fr(~rs),Server($S,~x) ]
    --[ServResp(K2s)]->
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid) , Out(<~cid, ~rs, c2,
   Xprime>)]
```

The let binding causes the modeled server to derive both K_1 and K_2 and produce a ciphertext to send. The premise of the rule requires an `In` to be seen on the public channel, and the existence of a server identity, along with the creation of x' and $\sim rs$. The conclusion of the rule instantiates a new type of persistent server fact called `!Server_2`, which contains the new data that the server has derived in this step. It also calls `Out` on the necessary items as in `Client_Hello`.

4.2.6 Client_Receive_Response

Rule `Client_Receive_Response` implements all of the logic after the second flow in Figure 4.1.

```
rule Client_Receive_Response:
    let
        D2=Xprime^~y
        K2c=kdf(D2,~rs)
    in
    [!Client($C,~y,Y,X,K1,~rc,~cid),In(<~cid, ~rs, c2, Xprime>)]
    --[ClientReceive(K2c)]->
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid)]
```

It uses the let construct to derive K_{C2} from a calculated D_{C2} using `kdf`, and then stores this derived key into a fact representing the new client state: `!Client_2`.

4.2.7 Client_Send_Data

The `Client_Send_Data` rule does not refer to an element of the key exchange outlined in Figure 4.1, but rather to the symmetric encryption that follows a successful key exchange according to the protocol.

```
rule Client_Send_Data:
    let
        clientmessage = 'clientmessage'
        ciphertext = aenc(K2c,clientmessage)
    in
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid)]
    --[CSD(ciphertext,clientmessage)]->
    [Out(ciphertext)]
```

This rule uses `K2c` (the client's resulting key after the exchange) to encrypt a plaintext created by the server using a symmetric encryption algorithm.

The premise of this rule is the `Client_2` fact with associated variables including the public identifier for the client and the session ID. The rule results in an `Out` fact which exposes the ciphertext. Additionally, the rule specifies an action fact `CSD(ciphertext, clientmessage)` which represents the fact that the client sent the ciphertext. The action fact is also associated with the original `clientmessage` plaintext.

4.2.8 Server_Send_Data

The `Server_Send_Data` rule is identical to the `Client_Send_Data` rule in function, except that it models the server sending encrypted data rather than the client.

```
rule Server_Send_Data:
    let
        servermessage = 'servermessage'
        ciphertext = aenc(K2s,message)
    in
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid)]
    --[SSD(ciphertext,servermessage)]->
    [Out(ciphertext)]
```

Notice that the premise takes in the !Server_2 fact, which has K2s (the server's key after the key exchange) in Figure 4.1 associated with it. The other difference is in the action fact, which has the name SSD to distinguish it from CSD.

4.2.9 Client_Rec_Data

The Client_Rec_Data rule models a situation that results from the server sending data symmetrically encrypted with K2s.

```
rule Client_Rec_Data:
    let
        plaintext_c = adec(K2c,ciphertext_c)
    in
    [!Client_2($C,~y,Y,X,K2c,~rc,~cid),In(ciphertext_c)]
    --[CRD(plaintext_c)]->
    []
```

It simply decrypts the ciphertext it receives, and puts CRD(plaintext_c) on the trace.

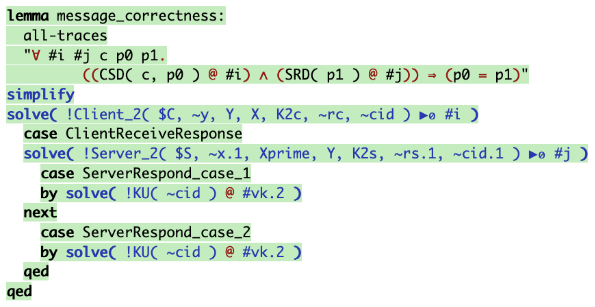
4.2.10 Server_Rec_Data

The Server_Rec_Data rule models the same scenario as Client_Rec_Data on the server side.

```
rule Server_Rec_Data:
    let
        plaintext_s = adec(K2s,ciphertext_s)
    in
    [!Server_2($S,~x,Xprime,Y,K2s,~rs,~cid),In(ciphertext_s)]
    --[SRD(plaintext_s)]->
    []
```

4.3 Restrictions

Restrictions unique_server and unique_client guarantee that the rule to create the client and server identities are only applied once, implying that there is at most one



```
lemma message_correctness:
  all-traces
  "∀ #i #j c p0 p1.
         ((CSD( c, p0 ) @ #i) ∧ (SRD( p1 ) @ #j)) ⇒ (p0 = p1)"
simplify
solve( !Client_2( $C, ~y, Y, X, K2c, ~rc, ~cid ) ▶₀ #i )
  case ClientReceiveResponse
  solve( !Server_2( $S, ~x.1, Xprime, Y, K2s, ~rs.1, ~cid.1 ) ▶₀ #j )
    case ServerRespond_case_1
    by solve( !KU( ~cid ) @ #vk.2 )
  next
    case ServerRespond_case_2
    by solve( !KU( ~cid ) @ #vk.2 )
  qed
qed
```

Figure 4.2: Tamarin result for `message_correctness`

client and server in the system state at any time.

```
1 restriction unique_server:
2     "All #i #j. CreateServer() @#i & CreateServer() @#j ==> #i = #j"
3 restriction unique_client:
4     "All #i #j. CreateClient() @#i & CreateClient() @#j ==> #i = #j"
```

We can examine line 2 to understand how this restriction is logically implemented. We state that for all moments in time `#i` and `#j`, where the rule for creating a server is applied at each of the two times, it is implied that the two times are equal. We will run a proof with and without these restrictions in place.

4.4 Lemmas

Now, we present the lemmas in the model.

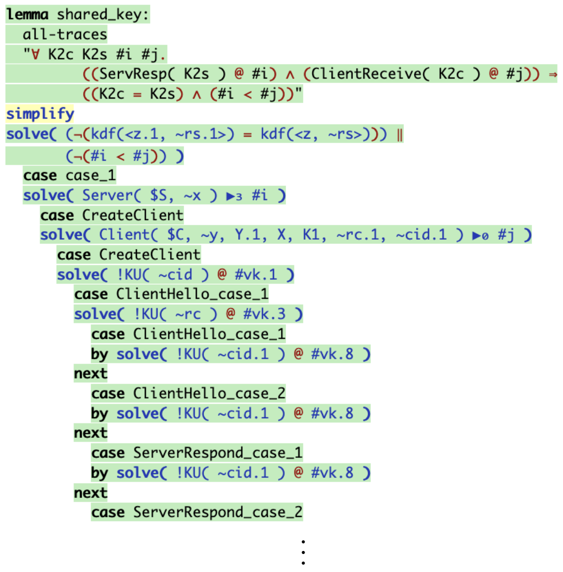
4.4.1 message_correctness

Lemma `message_correctness` requires that after the key exchange stage is complete, messages sent by a party will correctly decrypt to their original plaintexts.

```
lemma message_correctness:
    "All #i #j c p0 p1.
        CSD(c,p0) @ #i & SRD(p1) @ #j
        ==>
        p0 = p1
"
```

Specifically, we use the `CSD` action fact and the `SRD` action fact to model the instance where the client is sending data to the server, and we check that `p0` is equal to `p1`.

Figure 4.2 shows that Tamarin succeeds in finding a satisfying trace for the lemma.



```
lemma shared_key:
  all-traces
  "∀ K2c K2s #i #j.
         ((ServResp( K2s ) @ #i) ∧ (ClientReceive( K2c ) @ #j)) ⇒
         ((K2c = K2s) ∧ (#i < #j))"
simplify
solve( (¬(kdf(<z.1, ~rs.1>) = kdf(<z, ~rs>))) ‖
       (¬(#i < #j)) )
  case case_1
  solve( Server( $S, ~x ) ▶₃ #i )
    case CreateClient
    solve( Client( $C, ~y, Y.1, X, K1, ~rc.1, ~cid.1 ) ▶₀ #j )
      case CreateClient
      solve( !KU( ~cid ) @ #vk.1 )
        case ClientHello_case_1
        solve( !KU( ~rc ) @ #vk.3 )
          case ClientHello_case_1
          by solve( !KU( ~cid.1 ) @ #vk.8 )
        next
          case ClientHello_case_2
          by solve( !KU( ~cid.1 ) @ #vk.8 )
        next
          case ServerRespond_case_1
          by solve( !KU( ~cid.1 ) @ #vk.8 )
        next
          case ServerRespond_case_2
                         ⋮
```

Figure 4.3: Tamarin result for `shared_key`

4.4.2 shared_key

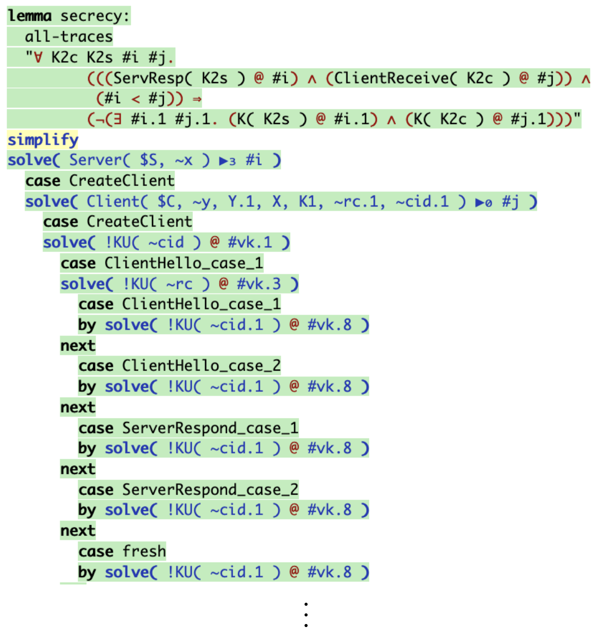
Lemma `shared_key` makes use of action facts `ServResp` and `ClientReceive` at times `#i` and `#j`.

```
lemma shared_key:
  "All K2c K2s #i #j.
      ServResp(K2s) @ #i & ClientReceive(K2c) @ #j
      ==>
      K2c = K2s & #i < #j
  "
```

The goal of this lemma is to prove that for every possible trace, when the `Server_Respond` rule is applied before the `Client_Receive_Response` rule (as should be the case), the two parties will have agreement on the shared key. Ideally, this will be true for an unbounded number of instantiations of each party. As we see in Figure 4.3 (truncated for space), the lemma is true in general. This result is identical in the case with the restrictions removed and the number of parties is unrestricted.

4.4.3 secrecy

The final lemma we prove is `secrecy`. This lemma states that for all traces where action facts `ServResp` and `ClientReceive` appear (in the correct order), there do not exist any points in time where the adversary knows the session key.



```
lemma secrecy:
  all-traces
  "∀ K2c K2s #i #j.
        (((ServResp( K2s ) @ #i) ∧ (ClientReceive( K2c ) @ #j)) ∧
         (#i < #j)) ⇒
        (¬(∃ #i.1 #j.1. (K( K2s ) @ #i.1) ∧ (K( K2c ) @ #j.1)))"
simplify
solve( Server( $S, ~x ) ▶₃ #i )
  case CreateClient
  solve( Client( $C, ~y, Y.1, X, K1, ~rc.1, ~cid.1 ) ▶₀ #j )
    case CreateClient
    solve( !KU( ~cid ) @ #vk.1 )
      case ClientHello_case_1
      solve( !KU( ~rc ) @ #vk.3 )
        case ClientHello_case_1
        by solve( !KU( ~cid.1 ) @ #vk.8 )
      next
        case ClientHello_case_2
        by solve( !KU( ~cid.1 ) @ #vk.8 )
      next
        case ServerRespond_case_1
        by solve( !KU( ~cid.1 ) @ #vk.8 )
      next
        case ServerRespond_case_2
        by solve( !KU( ~cid.1 ) @ #vk.8 )
      next
        case fresh
        by solve( !KU( ~cid.1 ) @ #vk.8 )
                         ⋮
```

Figure 4.4: Tamarin result for `secrecy`

```
lemma secrecy:
  "
    All K2c K2s #i #j.
    (
      ServResp(K2s) @ #i &
      ClientReceive(K2c) @ #j &
      #i < #j
    )
    ==> not(Ex #i #j . K(K2s) @ #i & K(K2c) @ #j)
  "
```

We can see in Figure 4.4 that Tamarin has proven this lemma to be true, indicating that SQUIC is indeed a secure protocol.

Chapter 5

Conclusion

In this thesis, we conducted a thorough exploration of two concepts in the field of cryptography: key exchange, and formal verification of cryptographic protocols. We introduced Tamarin Prover, describing in detail the components of a Tamarin model, and provided a minimal example to illustrate the use of rewrite rules in protocol specification. We then presented concepts in cryptography, providing insights into the general problem of key exchange. We presented Diffie-Hellman, a specific key exchange protocol, and analyzed it using Tamarin Prover. Finally, we presented and modeled SQUIC, a simplified version of QUIC. We proved in Tamarin that it has desired liveness and security properties while also demonstrating the practical utility of the formal verification method.

References

[1] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Computer Aided Verification* (N. Sharygina and H. Veith, eds.), (Berlin, Heidelberg), pp. 696–701, Springer Berlin Heidelberg, 2013.

[2] B. Blanchet, "The security protocol verifier ProVerif and its Horn clause resolution algorithm," in *9th Workshop on Horn Clauses for Verification and Synthesis (HCVS)* (G. W. Hamilton, T. Kahsai, and M. Proietti, eds.), vol. 373 of *EPTCS*, pp. 14–22, Open Publishing Association, Nov. 2022.

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

[4] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How Secure and Quick is QUIC? Provable Security and Performance Analyses," in *2015 IEEE Symposium on Security and Privacy*, pp. 214–231, 2015.

[5] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[6] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[7] P. Rogaway, "Authenticated-Encryption with Associated-Data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), p. 98–107, Association for Computing Machinery, 2002.

[8] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography.* 2015.

[9] S. Celi, J. Hoyland, D. Stebila, and T. Wiggers, "A tale of two models: Formal verification of KEMTLS via Tamarin," in *Computer Security – ESORICS 2022*

(V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, eds.), (Cham), pp. 63–83, Springer Nature Switzerland, 2022.