Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Simplified Language Extension for Tamarin-Prover

Diploma Thesis

2021
Bc. Jakub Šmahovský

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Simplified Language Extension for Tamarin-Prover

Diploma Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Martin Stanek, PhD.

Bratislava, 2021
Bc. Jakub Šmahovský

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Jakub Šmahovský
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Simplified Language Extension for Tamarin-Prover
Rozšírenie pre Tamarin-Prover so zjednodušeným jazykom

Anotácia: Tamarin-Prover je známy nástroj na overovanie vlastností bezpečnostných protokolov. Cieľom tejto práce je navrhnúť a implementovať rozšírenie pre Tamarin-Prover. Rozšírenie má umožniť jednoduchú definíciu klasických bezpečnostných protokolov a zrozumiteľne prezentovať čiastočné aj finálne výsledky, a tak zjednodušiť používanie a vyhýbať sa používateľským chybám. Ďalším cieľom je využiť vlastnosti klasických bezpečnostných protokolov na napomáhanie automatickému zastaveniu dokazovania.

Vedúci: doc. RNDr. Martin Stanek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 21.11.2019

Dátum schválenia: 21.11.2019 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Jakub Šmahovský
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Simplified Language Extension for Tamarin-Prover

Annotation: Tamarin-Prover is a well-known security protocol verification tool. The goal of this thesis is to design and implement an extension for Tamarin-Prover. The extension should improve usability and prevent user errors by allowing for simple description of classic security protocols, as well as presentation of intermediate and final results. An additional goal is to improve automatic proof termination by using properties of classic security protocols.

Supervisor:	doc. RNDr. Martin Stanek, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	21.11.2019

Approved: 21.11.2019 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank my supervisor, Martin Stanek. His guidance and insight were invaluable when writing this thesis.

Abstrakt

Práca umožňuje využívať základnú funkcionalitu nástroja nazývaného Tamarin-Prover pomocou zjednodušeného jazyka. Týmto spôsobom umožňujeme overovanie bezpečnostných vlastností kryptografických protokolov používateľom s obmedzenými znalosťami v tejto problematike. Uvádzame nový jazyk na špecifikáciu kryptografických protokolov. Tento jazyk je navrhnutý tak, aby bol zrozumiteľný a jednoducho sa používal. Implementujeme nástroj na preklad tohto jazyka do vstupu pre Tamarin-Prover a následnú prezentáciu výsledkov. Jazyk a implementácia sú navrhnuté tak, aby zamedzili najčastejším chybám neskúsených používateľov. Pri preklade berieme ohľad na efektivitu analýzy protokolov nástrojom Tamarin-Prover. Našu implementáciu demonštrujeme na vybraných protokoloch vrátane jedného súčasného protokolu z triedy protokolov Noise. Táto práca je určená predovšetkým pre začínajúcich záujemcov v oblasti kryptografických protokolov.

Kľúčové slová: kryptografické protokoly, Tamarin-Prover, prekladač

Abstract

This thesis allows users to utilize the basic functionality of the Tamarin-Prover tool using a simplified language. It allows verification of the security properties of cryptographic protocols by users with limited experience in this field. We introduce a new language for specification of cryptographic protocols. This language is designed to be easy to understand and use. We implement a tool that translates this language to the input of Tamarin-Prover and presents the results. The language and implementation are designed to avoid the most common mistakes of inexperienced users. The translation takes into account the efficiency of the Tamarin-Prover analysis. We demonstrate our implementation on chosen protocols including one modern protocol from the Noise protocol framework. This thesis is intended for beginners interested in the field of cryptographic protocols.

Keywords: cryptographic protocols, Tamarin-Prover, translator

Contents

Introduction

The goal of this thesis is to make formal verification of cryptographic protocols in the symbolic model more accessible to students and other users without professional experience in the field of cryptography. Cryptographic protocols are featured in a wide range of applications. It is crucial to provide reassurance that they achieve the desired security goals. In part, this can be done by automatic verification in the symbolic model. There are numerous tools that perform this kind of verification, one of the most recent ones being Tamarin-Prover [11, 14]. Tamarin-Prover features a powerful verification mechanism, but using this tool correctly requires a relatively high level of expertise. This may discourage some users or cause them to use the tool incorrectly.

We introduce a simplified language for interaction with Tamarin-Prover. The language was based on a project with similar goals, called VerifPal [9]. We implemented a tool that translates protocol specifications in this language to specifications accepted by Tamarin-Prover. The language is restrictive so that inconsistencies that do not appear in real-world protocols are forbidden. Inexperienced users are susceptible to various mistakes that can be avoided thanks to these restrictions. We aim to translate the output of Tamarin-Prover in a comprehensible way. This should help users to better understand the results of the analysis. The analysis of security properties of cryptographic protocols is in general an undecidable problem. We apply measures that increase the likelihood of termination in case of protocols specified in our language.

Chapter 1 provides an overview of the related topics. It primarily focuses on the features of Tamarin-Prover used in our own implementation. Chapter 2 focuses on our contributions. It begins with an overview of our input language. It also describes the process of discovering and avoiding user error, the translation of our input language to the input of Tamarin-Prover, and the reverse translation of Tamarin-Prover output. Chapter 3 concludes with several complete demonstrations of our implementation.

Chapter 1

Background

Many modern systems, applications, and devices require some form of communication. Few of the communication media and channels can be considered secure. In order to achieve confidentiality and integrity of the information in transfer, systems employ cryptographic functions such as encryption, hashing, or electronic signatures. Even if these functions themselves are sound, when used incorrectly, they do not provide the desired security properties. Cryptographic functions are often employed as parts of security protocols. The wide range of different applications requires a wide range of security protocols.

Weak security assurances, like the absence of known attacks, no longer provide sufficient guarantees for security protocols. For example, Lowe discovered an attack on the Needham-Schroeder protocol after over 15 years [10]. Proofs of security properties under formally specified conditions have since become the norm for newly introduced protocols. Proofs of the security properties offered by a security protocol are generally long and complex. Manually constructed proofs may contain flaws that remain unnoticed even after deployment of the protocol. Automatic construction of proofs is very challenging and manual proofs remain the main guarantees of security of modern protocols. At the same time, automated proofs of some parts of the protocol logic are a much welcome addition.

Proofs are constructed by first defining the protocol, the capabilities of an adversary, and the desired security properties in a strictly formal model. Then it is proven that the security properties hold in the protocol as long as the adversary does not exceed its capabilities. There are two main models used for such proofs, the computational and the symbolic model. Blanchet [2] provides a more comprehensive description of these models.

Messages in the computational model are represented as bitstrings and the adversary as a probabilistic Turing machine. The protocol is often represented as some form of a game where the success of the adversary represents an attack on the protocol.

Some limitations often apply to the adversary, for example, polynomial-time complexity. The aim is to prove that any adversary (with respect to the limitations) has a negligible probability of success. This is usually done by reduction to some underlying problem that is assumed to be hard (for example, the factorization of large numbers).

The symbolic model, generally attributed to Dolev and Yao [7], represents messages as terms in some term algebra. This means that base messages are represented by constant symbols or variables and complex messages are built over a signature containing function symbols. The function symbols represent cryptographic function applications. Operations on these messages are represented using message deduction rules that capture the properties of cryptographic functions. For example, the correctness of an encryption scheme must capture that decryption of an encrypted message using the correct key yields the original message. Message deduction rules also infer limitations on the adversary capabilities. The adversary is limited to use these rules in order to obtain new messages. For example, certain properties of a good hash function, like preimage resistance (see, for example, [1]), are represented by the absence of message deduction rules that would allow the adversary to obtain the original message from a hash. These limitations are often called the *perfect cryptography assumption*. Cryptographic functions are perfect, flawless, their underlying problems are impossible. A proof in the symbolic model is created by an exhaustive search through all the possible executions of a security protocol. As such the symbolic model proofs are much better suited for automated analysis.

In this thesis, we focus on the automatic construction of proofs in the symbolic model. More specifically, we attempt to make it accessible to users without professional experience in this field. We work with a tool for automatic proof construction called the Tamarin-Prover. We introduce a simplified language to specify security protocols and their underlying model. We use Tamarin-Prover to verify the security properties. We especially focus on preventing mistakes that users often make when writing protocol specifications.

1.1 Related work

Our work has been inspired by a project called VerifPal. Its aims and contributions are best described by quoting VerifPal's introductory paper, [9]:

> Verifpal is a new automated modeling framework and verifier for cryptographic protocols, optimized with heuristics for common-case protocol specifications, that aims to work better for real-world practitioners, students and engineers without sacrificing comprehensive formal verification features. Verifpal introduces a new, intuitive language for modeling proto-

cols that is easier to write and understand than the languages employed by existing tools. Its formal verification paradigm is also designed explicitly to provide protocol modeling that avoids user error.

VerifPal was designed with the aim of supporting protocol analysis using methods developed by the ProVerif tool [3]. The protocol is translated into abstract representation using Horn clauses. The verification process iteratively expands the knowledge of the adversary and checks whether some security properties have been violated until the adversary knowledge can no longer be expanded and a proof is reached.

In contrast to this approach, Tamarin-Prover [11, 14], another automated security protocol verification tool, represents the protocol by a multiset rewriting system as we discuss in section 1.3.1. It begins from a state that represents the violation of a security property and uses backwards search through the possible protocol executions in order to prove that such a state is unreachable.

We designed a framework with goals similar to those of VerifPal and use the Tamarin-Prover tool for verification of security protocols. This introduces another method of formal analysis to the idea of simplified protocol verification. We designed a language that bears resemblance to the VerifPal language and adopts many of its features. We implemented a tool that translates protocol specifications in this language to the input language of Tamarin-Prover. Our language has diverged from the VerifPal language in many ways. This allowed us to further simplify protocol specification and make it better suited for translation into Tamarin-Prover input. We present our input language as standalone and do not draw parallels to the VerifPal language to avoid misconceptions.

There was another project which featured translation of a simplified language to Tamarin-Prover input and should be mentioned. The bachelor thesis, Converting Alice & Bob Protocol Specifications to Tamarin, by Keller [8], features translation of the so-called “Alice and Bob notation” to Tamarin-Prover specification. This resulted in a way to specify security protocols in a simplistic manner using very few elements. We designed a protocol specification language that is much more verbose but provides a form of intuitive simplicity for real-world practitioners.

1.2 Term rewriting

Tamarin adopts methods originating from the area of term rewriting. In this section, we provide a brief summary of the notions from this area [11, 14]. We reduce the definitions to those that relate to our work.

An *order-sorted signature* $\mathbf{\Sigma}$ is a triple $(S, \leq, \Sigma)$ where S is a set of sorts, the relation $\leq$ is a partial order on S and Σ is a set of function symbols associated with

sorts such that the following two properties hold. First, for every $s \in S$, the connected component C of s in $(S, \leq)$ has a top sort $top(s)$ such that $c \leq top(s)$ for all $c \in C$. Second, for every function symbol $f : s_1 \times \ldots \times s_k \to s$ in Σ where $k \geq 1$, there is also a function symbol $f : top(s_1) \times \ldots \times top(s_k) \to top(s)$ in Σ. If S contains only one sort, we say that $\mathbf{\Sigma}$ is unsorted and write Σ instead of $\mathbf{\Sigma}$. We also denote the set of all k-ary function symbols in Σ by Σ^k.

We assume there are pairwise disjoint, countably infinite sets of variables $\mathcal{V}_s$ and constants $\mathcal{C}_s$ for all sorts $s \in S$. We denote the set of all variables by $\mathcal{V} = \cup_{s \in S} \mathcal{V}_s$ and the set of all constants by $\mathcal{C} = \cup_{s \in S} \mathcal{C}_s$. For arbitrary subsets $C \subseteq \mathcal{C}$ and $V \subseteq \mathcal{V}$ we denote the set of well-sorted terms constructed over $\Sigma \cup C \cup V$ by $\mathcal{T}_{\Sigma \cup C}(V)$.

Example 1. Cryptographic messages in the symbolic model are often represented by terms built over a signature. The signature contains function symbols that represent applications of cryptographic functions. For example, a constant p may represent some plaintext which in the computational model would be a bitstring. Similarly, a constant k may represent a symmetric key suitable for use in an encryption function. If the function symbol $enc \in \Sigma^2$ represents such an encryption function then $enc(p, k)$ may represent the encrypted message containing p encrypted using function enc with the key k.

Example 2. Sorts may be used to represent some system of types on the set of cryptographic messages. Plaintext p may be of some sort that identifies it as a plaintext, similarly k may be identified as a symmetric key and $enc(p, k)$ may be identified as a correctly encrypted message. At the same time, all these objects are of some top sort of all messages and there may be other inheritances, for example, a correctly encrypted message may still be a valid plaintext for another encryption. Our use of sorts (or rather the use of sorts within Tamarin-Prover) is much more limited and its details are discussed in section 1.3.

A *position* is a sequence of natural numbers. For a term t and a position p the notation $t|_p$ denotes the *subterm of t at position p* defined as

$$t|_p = \begin{cases} t & \text{if } p = [] \\ t_i|_{p'} & \text{if } p = [i] \cdot p' \text{ and } t = f(t_1, \ldots, t_k) \text{ where } 1 \leq i \leq k \\ \text{undefined} & \text{otherwise} \end{cases}$$

Position p is a *valid position of t* if $t|_p$ is defined. Term s is a *subterm of t* if $s = t|_p$ and p is a valid position of t. Additionally, if $s \neq t$ we say s is a *proper subterm* of t. A *ground term* is a term that contains no variables, i.e. none of its subterms are variables.

A *substitution* σ is a well-sorted function from $\mathcal{V}$ to $\mathcal{T}_{\Sigma \cup \mathcal{C}}(\mathcal{V})$ that corresponds to the identity function on all except for a finite set of variables. We identify σ with its

usual extension to an endomorphism on $\mathcal{T}_{\Sigma\cup\mathcal{C}}(\mathcal{V})$ and use the notation $t\sigma$ instead of $\sigma(t)$. Given terms t, s and a position p in t we denote by $t[s]_p$ the term resulting from replacing the subterm $t|_p$ in t by s.

Example 3. Positions and substitutions are used to describe manipulation with terms in a formal way. For example, the n-th of the topmost arguments of a term t can be denoted as $t|_{[n]}$. Also, using s as the n-th argument of t can be denoted as $t[s]_{[n]}$. We stick to the informal description if it is less confusing. For example, we write n-th argument of function f instead of writing $t|_{[n]}$ for every term t that is of form $f(...)$

A *rewriting rule* over a signature $\mathbf{\Sigma}$ is an ordered pair of terms (l, r) where $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$. We denote a rewriting rule (l, r) by $l \to r$. A *rewriting system* $\mathcal{R}$ is a set of rewriting rules. Given a rewriting system $\mathcal{R}$ we define the *rewriting relation* $\to_\mathcal{R}$ such that $s \to_\mathcal{R} t$ iff there is a position p in s, a rewriting rule $l \to r \in \mathcal{R}$ and a substitution σ such that $s|_p = l\sigma$ and $s[r\sigma]_p = t$. For example, given a rewriting rule $dec(enc(v, k)k) \to v$ the related pair $dec(enc(a(), b()), b()) \to_\mathcal{R} a()$ is correct according to a substitution σ such that $a()\sigma = v$ and $b()\sigma = k$.

A rewriting system $\mathcal{R}$ is *terminating* if there is no infinite sequence of terms $(t_i)_{i\in\mathbb{N}}$ with $t_i \to_\mathcal{R} t_{i+1}$. A rewriting system $\mathcal{R}$ is *confluent* if, for all terms t, s_1, s_2 with $t \to^*_\mathcal{R} s_1$ and $t \to^*_\mathcal{R} s_2$, there is a term t' with $s_1 \to^*_\mathcal{R} t'$ and $s_2 \to^*_\mathcal{R} t'$. A rewriting system $\mathcal{R}$ is *convergent* if it is terminating and confluent. The unique normal form of a term t with respect to a convergent rewriting relation $\mathcal{R}$ is the term t' such that $t \to^*_\mathcal{R} t'$ and there is no term t'' such that $t' \to_\mathcal{R} t''$. We denote this unique normal form by $t\downarrow_\mathcal{R}$.

An *equation* over a signature $\mathbf{\Sigma}$ is an unordered pair of terms $\{l, r\}$ where $l, r \in \mathcal{T}_\Sigma(\mathcal{V})$. We denote an equation $\{l, r\}$ by $l \simeq r$. An *equational presentation* is a pair $(\mathbf{\Sigma}, E)$ where $\mathbf{\Sigma}$ is a signature and E is a set of equations. Given an equational presentation $\mathcal{E} = (\mathbf{\Sigma}, E)$ we define the *equational theory* $=_\mathcal{E}$ as the smallest relation of equivalence on $\mathcal{T}_{\Sigma\cup\mathcal{C}}(\mathcal{V})$ containing all instances of equations in E.

Within this thesis, there are two very distinct uses of rewriting systems. First, *multiset rewriting systems* are used in section 1.3.1. A rewriting system may be specified such that it operates on terms that represent multisets of elements. The rules of such a rewriting system may be specified such that the order of entities within the original term is irrelevant. The formal definition of such rewriting systems are very detached from their intended use, so we proceed with the intuitive definition only. A multiset rewriting rule is a pair of multisets, left side and right side. Application of a multiset rewriting rule removes all the elements on the left side from the resulting multiset (with respect to a substitution) and adds all the elements on the right side to the resulting multiset. Second, we use rewriting in section 2.2.2 to decide equivalence within an equational theory. If the equations of an equational theory can be oriented to form a convergent rewriting system (i.e. an equation $l \simeq r$ is replaced by one of the rewriting

rules $l \to r$ or $r \to l$) then the question of equality of two terms can be reduced to the question of equality of their respective normal forms. We recommend consulting [6] for much more detailed information on both of these notions.

An $\mathcal{E}$-*matcher* of two terms, s and t, is a substitution σ such that $s =_{\mathcal{E}} t\sigma$. We use an $\mathcal{E}$-matching algorithm that verifies the existence of an $\mathcal{E}$-*matcher* of two terms in the translation of Tamarin output in section 2.4. For example, we may want to know if a term $h(enc(a(), b()))$ matches a template $h(x)$. Substituting the variable x for the term $enc(a(), b())$ in the template results in the original term, so it does.

1.3 Tamarin-Prover

Tamarin-Prover is a large project that supports a wide range of different features. It allows modeling unusual protocol constructions as well as very specific security properties while remaining quite efficient. Due to the restricted scope of our work, only a select part of the core features of tamarin are relevant to this thesis. In this section, we provide an overview of these features. For more detail, we direct the readers to the original Tamarin papers, [11, 14]. Tamarin-Prover has undergone many changes and improvements since it was first created. The current Tamarin-Prover Manual ([15]) sometimes contains more up-to-date information.

We describe the features of Tamarin-Prover in separate parts. First, we describe the security protocol model. The model is a semantical construction that represents the behavior of a security protocol and adversary using terms and multiset rewriting. Second, we describe how such a security protocol model can be specified in the input language of Tamarin-Prover. Much of the notation used to describe the theoretical model was carried over to the input language. Last, we describe how security properties can be specified using so-called trace formulas.

1.3.1 Security protocol model

Tamarin models cryptographic messages as terms over an order-sorted term algebra $(S, \leq, \Sigma \cup \mathsf{FN} \cup \mathsf{PN})$ with respect to an equational theory $=_{\mathcal{E}}$. The set of sorts S consists of a top sort *msg* and two incomparable subsorts, *fr* and *pub*, for fresh and public names. Formally, $S = \{msg,\ fr,\ pub\}$ and relation $\leq$ is the set $\{(fr, msg), (pub, msg)\}$.

There are countably infinite sets of fresh names FN and public names PN. Fresh names are used to model random values such as keys or nonces and public names are used to model known constants such as principal identities. Formally, fresh and public names are constant function symbols. We omit the empty parentheses in their notation, writing n instead of $n()$. We may prefix the name by a ~ symbol, indicating that the name is fresh or a $ symbol indicating that the name is public. We also use

the same notation to distinguish fresh and public variables, i.e. variables of sort *fr* and *pub* respectively. Enclosing a name in single quotes (e.g 'n' for a name n) indicates that the name is public and it originates from the protocol specification.

Tamarin allows the user to specify security protocols with messages constructed over an arbitrary set of unsorted function symbols Σ and an equational theory $=_{\mathcal{E}}$ formalizing the semantics of function symbols in Σ. The function symbols in Σ are used to represent cryptographic primitives that may be applied to messages such as encryption and decryption. The equations generating the equational theory $=_{\mathcal{E}}$ are used to represent the properties of cryptographic primitives such as the fact that decryption of an encrypted message using the correct key yields the original message.

Tamarin puts further restrictions on the set of equations generating the equational theory $=_{\mathcal{E}}$ in order to support verification of security protocols. For our work, it suffices to consider signatures and equational theories defined by built-in theories that follow these restrictions. The specific built-in theories used are discussed in more detail in section 2.3.2.

Cryptographic messages in the Tamarin-Prover model of a security protocol are the ground terms in $\mathcal{T}_{\Sigma \cup \mathsf{FN} \cup \mathsf{PN}}(\emptyset)$. Equality of terms is decided according to an equational theory $=_{\mathcal{E}}$.

Example 4. Let us consider a protocol that specifies a role Alice and uses symmetric encryption to encrypt and decrypt messages. In an instance of this protocol, the specific principal acting in the role of Alice may be modeled by an identity, by a public name $\$Alice$. The secret key used for encryption and the plaintext payload may be modeled by fresh names, $\sim K$ and $\sim p$. Formally, $Alice \in \mathsf{PN}$ and $K, p \in \mathsf{FN}$. To model the procedures used for symmetric encryption and decryption we may use binary function symbols enc, $dec \in \Sigma$. A message consisting of a plaintext $\sim p$ symmetrically encrypted with the key $\sim K$ is modeled by the term $enc(\sim p, \sim K)$. To model the properties of symmetric encryption we include the equation $dec(enc(x, y), y) \simeq_{\mathcal{E}} x$, where $x, y \in \mathcal{V}$, in the set of equations generating the equational theory $=_{\mathcal{E}}$.

Tamarin uses a labeled transition system with transitions defined by multiset rewriting to model the execution of a security protocol in the presence of an adversary. It extends the standard notion of multiset rewriting by support for creating fresh names, labeled rewriting rules, and persistent facts. Formal definitions of the model are quite complicated and not relevant to our work. We only provide informal definitions of the relevant notions and recommend examining the original papers for more details.

The state of the transition system is a multiset of so-called *facts*. Facts are built from terms over an unsorted fact signature $\Sigma_{\mathcal{F}}$ (different from the signature used to construct messages). Signature $\Sigma_{\mathcal{F}}$ contains a countably infinite number of function symbols called *fact symbols*. $\Sigma_{\mathcal{F}}$ is partitioned into *linear* and *persistent* fact symbols.

The set of all facts is the set $\{F(t_1, \ldots, t_k) | t_i \in \mathcal{T}_{\Sigma \cup \mathsf{FN} \cup \mathsf{PN}}(\mathcal{V}), 1 \leq i \leq k, F \in \Sigma^k_{\mathcal{F}}\}$. We extend the notion of persistence from fact symbols to facts. If a fact symbol F is persistent we also say that every fact $F(t_1, \ldots, t_n)$ is persistent (and similarly for linear fact symbols). Linear facts model resources that can only be consumed once. Persistent facts model inexhaustible resources that can be consumed an arbitrary number of times. For a clear distinction between linear and persistent facts we prefix persistent facts by an ! symbol.

Example 5. In order to initiate a session of a protocol, a principal may have to choose a symmetric key. We may decide to model this by a fact, e.g. *ChosenKey*(\$*Principal*, ~*Key*), where \$*Principal*, ~*Key* $\in \mathcal{V}$. If the fact *ChosenKey*(\$*Alice*,~*K*) can be established once then one session of the protocol may be executed with the principal \$*Alice* and the key ~*K*. However, we may want Alice to decide on a long-term key and allow her to run an unbounded number of sessions with this same key. In that case, we may model the fact that a principal has chosen a long-term key by a persistent fact, e.g. !*ChosenLongTerm*(~*Principal*, \$*LTKey*).

The transitions of the transition system are defined using labeled multiset rewriting. A *labeled multiset rewriting rule* is a triple (p, a, c) where p, a, c are multisets of facts. The facts in p are called the *premises*. The facts in a, representing the *labels* of the rule, are called the *actions*. And the facts in c are called the *conclusions*. A *labeled multiset rewriting system* is a set of labeled multiset rewriting rules. In order to be consistent with the input language of Tamarin-Prover, we will denote a rewriting rule (p, a, c) by $[p]$–$[a]$→$[c]$.

The support for creating fresh names is provided in the form of a special linear fact symbol $\mathsf{Fr} \in \Sigma^1_{\mathcal{F}}$. A fact $\mathsf{Fr}(n)$ denotes that the fresh name n is freshly generated. The only rule producing Fr facts (i.e. containing a Fr fact in its conclusions) is the special rule FRESH = []–[]→[$\mathsf{Fr}(x)$] where $x \in \mathcal{V}$.

The communication between the security protocol and the adversary (or the network which is considered the same assuming the standard Dolev-Yao adversary) is modeled by the special linear fact symbols $\mathsf{Out}, \mathsf{In} \in \Sigma^1_{\mathcal{F}}$. A fact $\mathsf{Out}(m)$ denotes that the message m was sent by the protocol and may be received by the adversary. A fact $\mathsf{In}(m)$ denotes that the message m is sent by the adversary and may be received by the protocol.

The knowledge of the adversary is modeled by the special persistent fact symbols $!\mathsf{K}$, $!\mathsf{KU}$, $!\mathsf{KD}$. The distinction between these facts is not relevant to our work and we recommend consulting the Tamarin-Prover manual ([15]) for details. For our purposes, it is sufficient to suppose that facts $!\mathsf{KU}(m)$ and $!\mathsf{KD}(m)$ have a similar meaning to the $!\mathsf{K}(m)$ fact. The $!\mathsf{K}(m)$ fact denotes that message m is known to the adversary. Additionally, we will only consider such multiset rewriting systems where a $!\mathsf{K}(m)$ action fact

is only introduced by the special predefined rewriting rule [!K(m)]−[!K(m)]→[In(m)].

Example 6. Continuing the previous example 5 we would like to model a protocol where a principal chooses a long-term key and then uses it to send out an arbitrary number of messages. We may use the rule [Fr(~K)]−[]→[!*ChosenLongTerm*($Principal, ~$K$)] to model the initialization and the rule [Fr(~p),!*ChosenLongTerm*(*Principal*, K)]−[]→[Out(*enc*(~p, K)] to model sending out messages. Note that ~K, $*Principal*, ~$p$, K, and *Principal* are all variables. Variables of sort *fr* and *pub* will only be instantiated to fresh and public names respectively.

An *execution* is an alternating sequence of states (multisets of facts) and multiset rewriting rule instances such that the following four conditions hold.

- The sequence begins with the empty multiset.
- Every step in the execution is valid with respect to the labeled multiset rewriting system.
- No two separate instances of the rewriting rule FRESH contain the same fresh name, i.e. the same fresh name is never generated twice.
- No rule in the multiset rewriting system except for the FRESH rule produces a Fr fact or creates a fresh name.

A *trace* of an execution is the sequence of sets of actions produced by the multiset rewriting rule instances within the execution.

1.3.2 Specification of security protocols

Tamarin limits the use of fresh and public names in a protocol specification. Given a set of function symbols Σ, with properties as discussed in the previous section, the set of terms that may be used within a protocol specification is $\mathcal{T}_{\Sigma \cup \mathsf{SN}}(\mathcal{V})$ where SN is a (countably infinite) subset of PN. We denote a constant $c \in \mathsf{SN}$ by 'c'. The set of variables $\mathcal{V}$ contains a countably infinite number of variables for all sorts *pub*, fr, and msg. Tamarin allows specification of variables, constants, and function symbols with syntax similar to the notation we have used in the previous text. In the following, we represent specification terms in a way that directly corresponds to mathematical notation. For example, we represent the fresh variable ~x by the string `~x` and the term *enc*('*const*', k) by the string `enc('const', k)`.

The fact symbols in $\Sigma_{\mathcal{F}}$ are specified beginning with an uppercase letter and may also contain underscores. We prefer the use of underscores to represent multiple-word fact symbols. For example, we prefer using the fact symbol !*Chosen_long_term* over the previously used !*ChosenLongTerm* and represent a fact !*Chosen_long_term*(P, K) by `!Chosen_long_term(P, K)`. We avoid using fact symbols that may be confused with the special fact symbols Fr, Out, In, !K, !KU, and !KD.

The multiset rewriting system of a security protocol model consists of predefined rules and so-called *protocol rules*. The predefined rules deal with fresh name generation and modeling the capabilities of a standard Dolev-Yao adversary. The protocol specification and additional adversary capabilities are specified using protocol rules. Tamarin puts further restrictions on protocol rules to ensure that the special fact symbols are used correctly. We only use such protocol rules that follow even stronger restrictions and we use the same restrictions in the definition of a *protocol rule*. For the original definitions (two slightly different definitions are used) we recommend consulting the original papers. A *protocol rule* is a multiset rewriting rule $[p]{-}[a]{\to}[c]$ following these restrictions:

- it only contains terms from $\mathcal{T}_{\Sigma\cup\mathsf{SN}}(\mathcal{V})$,
- it does not contain $\mathsf{!K}, \mathsf{!KU}$, and $\mathsf{!KD}$ facts,
- it does not contain the function symbol $*$ (denoting the multiplication of exponents),
- In and Fr facts only occur in premises,
- Out facts only occur in conclusions,
- the argument of a Fr fact is always a fresh variable,
- fresh and unsorted variables that appear in conclusions also appear in premises,
- premises only contain irreducible function symbols.

A *protocol* is a finite set of protocol rules. Some of the rules of a protocol are used to define additional capabilities of the adversary. We call these rules the *adversary rules*. We now define the syntax of protocol rule specification. In order to keep our definitions clear, we do not mention the possibility of adding or removing whitespace to improve readability. Most of the common programming language practices regarding whitespace also apply to the input language of Tamarin-Prover.

The Tamarin-Prover input language requires all rules to be named and allows specification of aliases before a rule. Rule names follow the same syntax as variable identifiers. An *alias* is an equation $v = t$ where $v \in \mathcal{V}$ and $t \in \mathcal{T}_{\Sigma\cup\mathsf{SN}}(\mathcal{V})$ specifying that all occurrences of the variable v should be replaced with the term t, therefore aliases are a purely syntactical construct. Aliases are non-recursive in the sense that a variable defined by an alias was not defined by any previous alias and is only used after the alias defining it. Aliases only apply to the rule before which they are specified.

A protocol rule $[p]{-}[a]{\to}[c]$ may be specified as

```
rule RN: let aliases in [ premises ] -- [ actions ] -> [ conclusions ]
```

where `RN` is replaced by the desired rule name. Word `aliases` is replaced by a list of aliases separated by line breaks. Words `premises`, `actions`, and `conclusions` are replaced by comma-separated lists of facts from p, a, and c respectively. Some terms

forming the right sides of aliases may be replaced within p, a, and c by the corresponding left-side variables. In case that no aliases are specified the following syntax is used.

```
rule RN: [ premises ] -- [ actions ] -> [ conclusions ]
```

Example 7. To demonstrate that the syntax of Tamarin-Prover input language is very similar to the notation we have used to describe the Tamarin model we specify the two rules from example 6 as follows.

```
rule Initialize: [Fr(~K)]--[]->[!ChosenLongTerm($Principal,~K)]
rule Send: [Fr(~p),!ChosenLongTerm(Principal,K)]--[]->[Out(enc(~p,K)]
```

For a more involved example let us consider the following protocol. The protocol is executed by two principals, Alice and Bob, who share a symmetric key K. Alice generates a payload p, encrypts it using the key K, and sends it to Bob. Bob declares that the protocol succeeded if he successfully decrypts a payload p using the shared key K. We may specify this protocol as follows.

```
rule init: [
  Fr(~K)
]-->[
  Alice_init($A, $B, ~K),
  Bob_init($B, $A, ~K)
]

rule Alice: [
  Alice_init(A, B, K),
  Fr(~p)
]-->[
  Out(enc(~p, K))
]

rule Bob:
let
  p = dec(m, k)
in [
  Bob_init(B, A, K),
  In(m)
]--[
  Success(p)
]->[]
```

1.3.3 Trace Formulas

Security properties of a protocol model in Tamarin are specified as properties of the traces of its executions. For this purpose Tamarin introduces the sort *temp* for timepoints and variables of sort *temp* are called *temporal variables*. Note that timepoints

and temporal variables are never used to construct messages and the *temp* sort is completely distinct from the sorts used to define the message signature. To clearly distinguish temporal variables we prefix them with a # symbol. Timepoints correspond to the indices of the sets of actions within a trace of an execution of a protocol.

A *trace atom* is one of the following

- $l = r$; where $l, r \in \mathcal{T}_{\Sigma \cup \mathsf{SN}}(\mathcal{V}_{msg})$
- $\#i < \#j$; where $\#i, \#j \in \mathcal{V}_{temp}$
- $\#i = \#j$; where $\#i, \#j \in \mathcal{V}_{temp}$
- $F@\#i$; where $\#i \in \mathcal{V}_{temp}$ and F is a fact built over $\Sigma_{\mathcal{F}}$ and terms from $\mathcal{T}_{\Sigma \cup \mathsf{SN}}(\mathcal{V}_{msg})$

and a *trace formula* is a first order formula built over trace atom (quantified over both messages and timepoints).

A *valuation* θ is a function assigning messages to message variables and timepoints to temporal variables, i.e. it respects sorts *msg* and *temp*. Variables of sort *fr* and *pub* are not used in property specification. The definition of a valuation is naturally extended to terms and facts built over variables from $\mathcal{V}_{msg}$. We say a trace tr together with a valuation θ *satisfies* a trace atom a if one of the following holds:

- a is a term equality atom $l = r$ and $\theta(l) =_{\mathcal{E}} \theta(r)$
- a is a timepoint ordering atom $\#i < \#j$ and timepoint $\theta(\#i)$ precedes the timepoint $\theta(\#j)$ in trace tr (i.e. corresponds to a smaller index)
- a is a timepoint equality atom $\#i = \#j$ and $\theta(\#i) = \theta(\#j)$ (i.e. θ maps both temporal variables $\#i$ and $\#j$ to the same timepoint)
- a is an action atom $F@\#i$ and the i-th set of actions in trace tr contains the fact F (or a fact equivalent to F under the equational theory $=_{\mathcal{E}}$)

The satisfaction of complex formulas is defined inductively by the usual definitions of satisfaction of logical operators and quantifiers.

Trace atoms are specified using the same syntax as in the previous mathematical notation. For logical operators and quantifiers, the following are used: `not` for negation, `&` for conjunction, `|` for disjunction, `==>` for implication, `Ex` for existential quantification, and `All` for universal quantification. The quantification is separated from a formula by a `.` symbol. Additionally, the specification needs to define whether Tamarin-Prover should search for a trace that satisfies the formula or if it should prove that all traces satisfy it (by searching for a trace that does not). This is done by quantification over traces using keywords `exists-trace` and `all-traces` followed by the formula enclosed in quotation marks. Finally, Tamarin calls formulas specifying security properties *lemmas*, it requires that the user provides a name for every lemma and a lemma is introduced using the `lemma` keyword. A trace property consisting of a formula `F`, quantified over all traces and with the name of `LN` can be specified as

```
lemma LN: all-traces "F"
```

Example 8. Let us consider a protocol that facilitates the communication of two principals, Alice and Bob. We would like to verify the confidentiality of the message payload sent from Alice to Bob. We may do this by first adding an action fact to the rule that represents the payload `p` being received by Bob. We will choose an action fact `Confidential(p)`. This fact will represent that if Alice and Bob got to that point in the execution of the protocol with the message payload `p` then we suppose that the adversary will never be able to learn `p`. For example, a reasonable point to place the `Confidential(p)` action in example 7 is alongside the `Success(p)` action (or replacing it). We may specify our confidentiality supposition by the following trace property.

```
   All p #i #j. Confidential(p) @#i ==> not K(p) @#j
```

Adding additional parts of a protocol may require changing the formula. For example, we may want to add a standard adversary capability that allows it to act as a principal in one of the roles. This is often done in Tamarin by creating a rule that reveals the long-term values of a principal to the adversary and marks this principal as dishonest. The formula then needs to specify, that the adversary does not know `p` as long as neither of the principals exchanging `p` is dishonest. We may choose to mark a principal `P` as dishonest using a `Dishonest(P)` fact in the rule revealing long-term values. We may then choose to bind principal identities `A` and `B` together with the specific payload `p` using a fact `Instance(A, B, p)` in some initiation rule, i.e. a rule where it is decided that Alice `A` will be sending payload `p` to Bob `B`. Finally, we may change the trace property as follows.

```
All A B p #ins #con #adv.
    Instance(A, B, p) @#ins &
    Confidential(p) @#con &
    not (Ex #dis. Dishonest(A) @#dis) &
    not (Ex #dis. Dishonest(B) @#dis)
  ==>
    not K(p) @#adv
```

Or in a somewhat simpler form without negation:

```
All A B p #ins #con #adv.
    Instance(A, B, p) @#ins &
    Confidential(p) @#con &
    K(p) @#adv
  ==>
    (Ex #dis. Dishonest(A) @#dis) |
    (Ex #dis. Dishonest(B) @#dis)
```

There is another feature utilizing trace formulas that we use in a very limited way. Tamarin allows users to specify restrictions that limit the set of traces to be considered in protocol analysis. We use one specific, standard restriction to enforce equality of two terms. The exact specification of this restriction is as follows.

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

Specifying this restriction causes all rules that use action facts `Eq(l, r)` to be instantiated such that terms l and r are equal with respect to $=_{\mathcal{E}}$.

Chapter 2

Our language and compiler

We attempt to provide a simpler way to interact with some of the core features of Tamarin-Prover. We allow the user to specify security protocols in a simplified specification language. This limits the protocols to some of the more standard cryptographic primitives. The specification of security properties is also limited to predefined templates. We have designed this language with the goal of making it intuitive for first-time users, explicit, and resistant to user error. Relevant assumptions about a protocol are explicitly stated in the specification and many of the common mistakes are captured by the compiler.

In this chapter, we provide a description of our language. We describe the sanity checks aimed at avoiding user errors including how we implement them. We describe the translation of protocol specifications in our language to Tamarin. Finally, we describe the implementation of the reverse translation of the output produced by Tamarin-Prover.

2.1 Input language

We begin with an example to demonstrate most of the syntax of our input language. It also shows which details of a protocol can be covered by the analysis and which details are abstracted from.

Example 9. We will specify a protocol that shows a single Diffie-Hellman exchange. A Diffie-Hellman exchange allows principals to establish a shared secret using asymmetric keypairs based on elements of a cyclic group. The group has to be chosen such that the discrete logarithm problem is hard. In our model, the discrete logarithm problem is considered impossible due to the perfect cryptography assumption. In real applications, operations on cyclic groups cannot be performed directly on messages. There needs to be a mapping between the space of available messages and the elements of the chosen cyclic group. In our model, this is also abstracted from. A term represents both a

viable message and the corresponding element in any cyclic group.

The protocol is performed by two principals, Alice and Bob. They perform a Diffie-Hellman exchange between Alice's static keypair and Bob's ephemeral keypair. We assume that Bob has prior knowledge of Alice's public key. Bob uses the shared secret to send an encrypted message to Alice.

The initial assumptions state that Alice generated a private key a_s and distributed the public key a_p, constructed as g^{a_s}, to Bob. We define these assumptions as follows:

```
principals: Alice, Bob
Alice [
  knows private as
  distributed ap = 'g'^as
]
Bob [
  knows public ap
]
```

In a protocol session, Bob generates his private key b_s and constructs his public key b_p as g^{b_s}. He constructs the symmetric key s as $a_p^{b_s}$ and the ciphertext c of the message m. He sends both the ciphertext c and his public hey b_p to Alice. Alice also constructs s, but as $b_p^{a_s}$. This should be equal to Bob's s because $b_p^{a_s} = g^{b_s * a_s} = g^{a_s * b_s} = a_p^{b_s}$. Alice uses s to decrypt the ciphertext c and learn m. We specify this protocol as follows:

```
Bob [
  generates bs, m
  bp = 'g'^bs
  s = ap^bs
  c = ENC(s, m)
]
Bob -> Alice: bp, c
Alice [
  s = bp^as
  m = DEC(s, c)
]
```

The obvious security property of this protocol that needs to be verified is the confidentiality of the private message m. We need to realize that the protocol specification actually contains two messages specified as `m`. One was generated by Bob and the other was deconstructed from a ciphertext by Alice. These two messages should be equal if the protocol performs correctly, but they may be different if the adversary replaces the ciphertext `c` in transit. This is also the reason why we say Alice's `s` *should* be the same as Bob's, but it cannot be guaranteed. We specify the security properties of both messages `m` as follows:

```
  queries [
    confidentiality? Bob's m
```

```
    confidentiality? Alice's m
]
```

When running the protocol we learn that the first property holds while the second one does not. This was expected. The first property is the desired security property of the original protocol. The second property does not hold. The adversary may send Alice his own messages just like Bob would.

2.1.1 Messages

Messages in our language are terms constructed using predefined function symbols. The list of function symbols and their semantics are described in section 2.1.2. Atomic messages are constants and variables.

Constants correspond to the names from SN in Tamarin (also called constants in the Tamarin-Prover manual [15]). In our language, a constant is specified by an identifier enclosed in single quotes, for example, a constant c is specified as `'c'`. Constants are public values that are unchanging.

For variables, we introduce the notion of scopes into protocol specification. Variables are specified by a variable identifier and a scope identifier. Scopes represent the knowledge of individual principals. Scope identifiers are the variable identifiers used for principals. For example, Alice's variable v is specified as `Alice's v`. Explicitly stating the scope is only necessary when defining security properties. Within the protocol specification, the scope is clear from the context and a variable is specified only by the variable identifier. For example, Alice's variable v is specified just as `v` in the protocol specification. Scopes allow us to specify protocols in a very natural way. In example 9, we discussed that `m` may be different for each principal. Specifically, there are two separate variables `m`, but they are expected to be equal. Sometimes protocols may use the same identifier for variables that are expected to be different. For example, two principals may both generate a nonce and exchange it with the other principal. Each may use the identifiers `N` for their own nonce and `No` for the nonce they received. We consider this use of scopes confusing and avoid using it, but we do not discourage it (even though our compiler can distinguish it).

2.1.2 Cryptographic functions

We do not allow users to define their own function symbols unlike many protocol verifiers including Tamarin. Instead, we provide a predefined set of function symbols that can be used to represent the most common cryptographic functions. This follows the idea presented by VerifPal that custom primitives may lead to user error ([9]). The syntax of most of the cryptographic functions was adopted from VerfiPal. Some of the

functions are not supported by VerifPal. In those cases, we try to make our syntax similar.

Function symbols follow the perfect cryptography assumption. This means that terms can only be constructed and deconstructed using the function symbols and equalities mentioned here. In example 9, we mentioned that the discrete logarithm problem is impossible. There is no function symbol or equality that allows recovering the exponent from an exponentiation term. This makes the discrete logarithm problem impossible in our model. This notion applies to all cryptographic schemes and the problems they are based on.

A tuple is the simplest cryptographic function. They represent a simple concatenation of terms. We allow denoting tuples as a list of comma-separated terms enclosed in curly brackets. This syntax is similar to the one used by the so-called Alice and Bob notation. For example, a triple of the terms `m1`, `m2`, and `'const'` is specified using `{m1, m2, 'const'}`.

In verifiers as well as programming languages, tuple deconstruction is often done using special functions that retrieve a single element at a time or a new tuple of a range of elements. We consider this method of deconstructing tuples confusing and awkward to use. Instead, we support a special syntax to spread tuples. For example, if `t` is a variable representing a tuple `{m1, m2, 'const'}`, then it can be deconstructed using the assignment statement `{x1, x2, x3} = t`, which assigns `m1` to `x1`, etc.

We use binary function symbols `ENC` and `DEC` to represent symmetric encryption and decryption respectively. The term `DEC(key, ENC(key, message))` is equivalent to `message`.

From asymmetric cryptography we support basic encryption and signing. We use the unary function symbol `PK` to represent public key derivation. Given a secret key `sk`, the term `PK(sk)` is the corresponding public key. We use binary function symbols `AENC` and `ADEC` to represent asymmetric encryption and decryption respectively. The term `ADEC(sk, AENC(PK(sk), message))` is equivalent to `message`. We use binary function symbol `SIGN` to represent signing. A signature is verified using a check `SIGNVERIF` (checks are described in 2.1.3). The following check is successful:

```
SIGNVERIF(PK(sk), message, SIGN(sk, message))?
```

We use the unary function symbol `HASH` to represent hashing. We do not support any other hash functions. To model hash functions with arity greater than one, the user can use a tuple in the unary hash function, for example, `HASH({x1, ..., xn})`. To model different hash functions the user can manipulate the first value in the tuple. For example, `HASH({'0', value})` may be used to represent the default hash function and `HASH({'1', value})` may be used to represent a different hash function. To model keyed hash functions (HMAC or key derivation functions) the user can also just

use the key as one of the elements of the tuple. For example, a MAC of a message `message` using symmetric key `key` may be created as `HASH({key, message})`.

We use the binary function symbol `^` to represent exponentiation in a cyclic group. We denote the application of `^` in infix notation (for example, `x^y`). Following the conventions of Tamarin ([15]) and contrary to the usual interpretation we interpret exponentiation as left-associative. For example, `x^y^z` is the same as `(x^y)^z` because in cryptography `(x^y)^z` is much more common. Tamarin's built-in theory for cyclic groups also supports multiplication and inverses in a limited way. We use this theory to verify protocols that use exponentiation, but do not yet support the other operations. The semantics of these operations are formalized by the following equations (where the asterisk ($*$) denotes multiplication):

$$(x\hat{}y)\hat{}z \simeq x\hat{}(y*z) \qquad x\hat{}1 \simeq x \qquad x*y \simeq y*x$$
$$(x*y)*z \simeq x*(y*z) \qquad x*1 \simeq x \qquad x*x^{-1} \simeq 1$$

2.1.3 Protocol specification

A protocol specification in our language contains initial assumptions and the sequence of actions in a single protocol session. The idea is to specify the actions in the order in which they happen in a real execution of the protocol. Often the exact order is not relevant to the security of a protocol. For example, if decryption fails, then the adversary only leans that it failed because the next message does not arrive. It does not matter if the actions between a failed action and the next message get executed because their results are just discarded anyways. Still, we recommend keeping the order of actions consistent.

All assumptions and actions in a protocol are specified as statements. All statements, except for message statements, are grouped into principal blocks. The initial assumptions represent some actions that happened before the protocol session may begin. It makes sense to specify them at the beginning of the protocol specification. We recall the specification of initial assumptions from example 9 for demonstration:

```
1  Alice [
2    knows private as
3    distributed ap = 'g'^as
4  ]
5  Bob [
6    knows public ap
7  ]
```

There are 2 principal blocks, one for `Alice` on lines 1-4 and one for `Bob` on lines 5-7. There are 3 statements on lines 2, 3, and 6. A block specifies the scope of all variables within. For example, variable `ap` in the block of `Bob` is `Bob's ap`. Principals may

only use or send variables that they know – variables they have previously declared or received.

The first declaration on line 2 is a `knows` statement with the modifier `private`. This statement declares knowledge of a long-term private variable `as`. Private variables are guaranteed to always be instantiated to different values that are also distinct from public values and constants. The `knows private` statement also allows declaration of pre-shared secret. If several principals use this statement with the same variable identifier, they will all know the same long-term private variable.

The `knows` statement on line 6 uses the modifier `public`. This statement declares knowledge of a long-term public variable `ap`. Public variables are variables that the adversary always knows, but principals have to declare that they know them. If a public variable with the same identifier was declared before, then this statement declares knowledge of the same variable. Otherwise, this statement declares a new public variable. Separate public variables are expected to be distinct under normal circumstances, but they do not have to be. Several public variables may be instantiated to the same value or even to a constant.

The `distributed` statement, like the one on line 3, declares that the distribution of a public variable occurred before the session was initiated. A `distributed` statement contains the variable that was distributed as well as a term that describes how it was constructed. The statement on line 3 declares a public variable `ap` that was constructed as `'g'^as`. The `distributed` statement can be used to declare any public variable, that is constructible before the first protocol session, but it is most suitable for the declaration of public keys.

The principals themselves specify public variables that represent their identities. Variables `Alice` and `Bob` are the identities of Alice and Bob, but they were not used in this protocol.

The remaining statements specify the protocol session itself. We recall the rest of the protocol specification from example (9) for demonstration:

```
1  Bob [
2    generates bs, m
3    bp = 'g'^bs
4    s = ap^bs
5    c = ENC(s, m)
6  ]
7  Bob -> Alice: bp, c
8  Alice [
9    s = bp^as
10   m = DEC(s, c)
11 ]
```

The first statement on line 2 is a `generates` statement. It declares ephemeral pri-

vate variables `bs` and `m`. The `generates` statement represents random generation of ephemeral private values. Just like private variables from `knows private`, the variables from `generates` are always instantiated to distinct values. Additionally, the same variable from `generates` will also be instantiated to different values in separate protocol sessions. In our example, if Alice and Bob run the same protocol instance twice, then Bob will use two different values for `bs`, but Alice will use the same long-term key `as`.

The other statements in blocks are assignments. Assignments declare ephemeral private variables. These variables are always instantiated to the same value as the corresponding term on the right side of the assignment. Assignments support a special syntax, that allows using tuples on their left side in order to deconstruct a tuple on the right side. For example, the following two assignments correctly construct and deconstruct nested tuples:

```
t = {{m1, m2}, 'const'}
{{x1, x2}, 'const'} = t
```

Assignments may use constants or variables that are already known on the left side. This does not reassign the variables but instead performs an implicit equality assertion. For example, in the lines above, the principal verifies that the last element of `t` is equal to `'const'`. This can be very convenient when principals share some values that should be included in messages. The value in the message can be checked without assigning it to a variable with a new identifier.

Line 7 contains a message statement. It is a message sent from `Bob` to `Alice` containing messages `bp` and `c`. A message in transit could be changed by the adversary. This means that the message received cannot be represented by the same variable as the message sent. The variable identifier stays the same, but the scope changes, resulting in two separate variables. Scope identifiers are not used because the scope is clear from the identifiers of the sender and the recipient. In our example, `Bob` sends `Bob's bp` and `Bob's c` and `Alice` receives `Alice's bp` and `Alice's c`. Tuples may also be used in messages. They do not affect the security of the protocol but may help organize messages. If principals receive variables that they already know, then they perform an implicit equality assertion just like in assignments.

The remaining statements are called checks and they did not occur in example 9. Checks represent assertions that need to succeed in order to continue the protocol. A check is performed by a single principal and it is also specified in a principal block. If it fails, then the other principals are unaffected. Checks are specified as methods with arguments. We add a question mark after them to emphasize that we are making an assertion. There are three types of checks – unary and binary `EQUALS`, and `SIGNVERIF`.

The unary `EQUALS` check should be used after an implicit equality assertion in a message statement or an assignment. For example, if an assignment contains an

already known variable `v` on the left side, then it should be followed by

```
EQUALS(v)?
```

If an implicit equality assertion does not have a matching unary `EQUALS` check before the end of a block, then the compiler prints out an info message to warn the user. These messages are meant to inform the user about accidental unwanted assumptions in the protocol, but they can be turned off using a declaration

```
unary-equals: implicit
```

Declarations are discussed in section 2.1.6.

The binary `EQUALS` check simply verifies the equality of its arguments. For example, the equality of `mac` and `{HASH(k, m)}` can be verified as

```
EQUALS(mac, HASH({k, m}))?
```

The `SIGNVERIF` check is used to verify a signature. For example, if message `m` was signed with a secret key `sk`, `pk` is the public key corresponding to `sk` and it created a signature `s` then the following check succeeds:

```
SIGNVERIF(pk, m, s)?
```

If we substitute the full terms for the variables we get

```
SIGNVERIF(PK(sk), m, SIGN(sk, m))?
```

2.1.4 The adversary model

The capabilities of the adversary and the security properties are closely related. Before we describe how to specify the security properties of a protocol, we have to define the adversary model. The security properties are specified such that they put limitations on some of the capabilities of the adversary.

By default, Tamarin provides predefined rules modeling the so-called Dolev-Yao adversary [7]. We extend the capabilities of this adversary by long-term reveals, which is a very common extension. We describe the capabilities of such adversary using notions of our input language as follows:

1. The adversary has knowledge of all public variables, i.e. principal identities, variables declared using `knows public`, variables declared using `distributes`, and constants.
2. The adversary can deduce new messages from messages in its knowledge using function symbols *relevant to the protocol* and with respect to the equivalences stated in 2.1.2.
3. The adversary has full control over the network. He is able to intercept, stop, replace, and produce messages that are sent or received by the protocol principals.

4. The adversary can gain knowledge of all long-term private messages of any protocol principal. This action is called *long-term reveal* and the principal is considered *dishonest* afterwards.

Function symbols *relevant to the protocol* that are mentioned in point 2 are all the function symbols that appear directly in the protocol specification or they are related to such function symbols by some equivalences stated in 2.1.2. This just means that we do not include completely unrelated function symbols in protocol verification for obvious reasons. For example, if the protocol does not use operations on cyclic groups then there is no reason why the adversary should be able to use them. At the same time, if the protocol uses symmetric encryption, but no symmetric decryption, then the adversary can still decrypt if he knows the correct key. This does not change the capabilities of the adversary in any meaningful way, however, the definition would be incomplete without it.

Points 1 through 3 represent the capabilities of the standard Dolev-Yao adversary. Point 4 represents the extension of adversary capabilities by long-term reveals. Long-term reveals allow the adversary to act as one or more of the principals. This has two possible meanings. It often means that the adversary really is a proper principal in some protocol instances. In real-world applications, some of the actors performing the protocol may be malicious. Otherwise, it means that the adversary has been able to compromise another principal. The compromised values are long-term because it is most plausible to compromise values that are not normally discarded after a protocol session. It is often reasonable to consider the possibility of compromising some subsets of ephemeral values, but we do not support it yet.

When specifying security properties, we limit the capability to perform long-term reveals. We verify that a property holds in a protocol instance where all principals are honest (or at least up to some point). However, the adversary may run other instances of the same protocol in the role of a proper principal. To give an example, we briefly discuss how instantiated variables are denoted in Tamarin (and our) output. When a variable is instantiated to a (private or public) name, the variable identifier is kept and it is suffixed by a dot and a number to distinguish between the various instantiations of variables.

Example 10. In example 9, one instance may be performed by `Alice.1` in role of `Alice` and `Bob.1` in role of `Bob`. We want to find an attack against this instance. As part of this attack, the adversary may perform a long-term reveal on `Bob.2`, making `Bob.2` dishonest. He may even execute the protocol as `Bob.2` in role of `Alice` together with `Alice.1` in role of `Bob` and it does not invalidate the attack. However, he cannot perform a long-term reveal on `Alice.1` or `Bob.1`.

So far this is the only adversary model supported by our program. We chose this

model because it is one of the adversary models that are often considered in textbook attacks. At the same time, it does not attempt to capture any of the possible flaws in cryptographic primitives. Implementing the support for other adversary models would certainly be meaningful in the future.

2.1.5 Specification of security properties

Security properties are specified as *queries*, i.e. suppositions to be verified. We support the specification of 4 types of queries – confidentiality, forward secrecy, authentication, and injective authentication. In this section, we describe the syntax and semantics of each of these queries and how they are parametrized. All queries are specified within a `queries` block after a protocol specification. The syntax of this block is similar to the syntax of principal blocks and it was demonstrated at the end of example 9.

Security properties often work with some important points in the execution of a protocol. The order in which statements in a protocol are executed is not always relevant to security properties. Only the order with regard to actions that have side effects matters. The only actions that have side effects are sending out messages and finishing the protocol. Sent messages can be observed by the adversary and finishing the protocol means accepting the values learned from the protocol as valid (authentic, confidential, etc.). An important point in a protocol execution is when a principal *commits* to executing a block. This also includes the execution of subsequent blocks up until the next message from this principal or the end of the protocol. A principal commits to a block if (1) he has already executed the previous blocks (and sent messages), (2) he has received all the necessary messages since the last block, and (3) all message constructions/deconstructions, checks and assertions in the block would be successful. In a real-world execution, this would simply mean that the principal (1 and 2) is supposed to execute the block right now, and (3) tries to execute the block. If some assertion fails, then he stops the execution before any action that has side effects happens. Note that in a real-world execution any action may have side effects, especially computation time. This leaves room for side-channel attacks which are not covered by our model.

The confidentiality query verifies the basic supposition that the adversary is unable to obtain some variable. It is parametrized by a scope-and-variable identifier. For a variable `Principal's variable` the confidentiality query is specified as

```
confidentiality? Principal's variable
```

The confidentiality query of `Principal's variable` states that if (1) `Principal` commits to executing the first block where he knows `variable` and (2) he and all the other principals within the same protocol instance are honest, then the adversary cannot learn `Principal's variable`.

The forward secrecy query provides a stronger alternative to the confidentiality query. It is once again parametrized by a variable specified in its full form. For a variable `Principal's variable` the forward secrecy query is specified as

```
forward-secrecy? Principal's variable
```

The forward secrecy of `Principal's variable` states that if (1) `Principal` commits to executing the first block where he knows `variable` and (2) he and all the other principals within the same protocol instance are honest *until* `Principal` executes said block, then the adversary cannot learn `Principal's variable`.

This property specifies that even if the adversary learns all long-term values of the principals performing the protocol after the transfer happens, the variable in question would remain confidential. This query offers, in our opinion, a good baseline for forward secrecy properties. However, most protocols today provide confidentiality guarantees even if some combinations of the ephemeral variables of a protocol are revealed by the adversary. For example, even the old attack on the Needham-Schroeder protocol by [10] requires compromising an ephemeral variable.

The authentication query verifies that if a principal believes he received a message from another principal, then this message was indeed sent by that principal. It is parametrized by the sender, the recipient, and the message variable. For a message form `Sender` to `Recipient` containing `variable` the authentication query is specified as

```
authentication? Sender -> Recipient: variable
```

This property states that if (1) `Recipient` commits to executing the block directly after he receives `variable` and (2) `Sender`, `Recipient`, and all other principals in the same protocol instance are honest, then `Sender` must have sent `variable` (at any time).

Example 11. Consider a protocol with two principals, `Sender` and `Recipient`, an instance of this protocol with `Sender.1` in role of `Sender` and `Recipient.1` in role of `Recipient`. We may want to verify the following query

```
authentication? Sender -> Recipient: variable
```

This query verifies that when `Recipient.1` believes that he received `variable.1` from `Sender.1` then `variable.1` was indeed sent by `Sender.1` and not any other `Sender` (or anyone else). It does not verify whether `Sender.1` sent it in the same session as `Recipient.1` received it. It also does not verify if `Sender.1` intended to send it specifically to `Recipient.1`.

The injective authentication query is a stronger version of the authentication query. Its goal is to capture some replay attacks. If the messages of a protocol do not contain enough ephemeral information, then they may be replayed again by the adversary at a

later time. The recipient accepts the same authentication message more than once. The injective authentication query specifies that a message received in a certain protocol session must have also been sent during that session. The beginning of a session is at some point before the first message sent by any of the principals. It is also the same for every principal in the same session. This means that a session may start for the responder in a protocol before he can know about it. For a message form `Sender` to `Recipient` containing `variable` the injective authentication query is specified as follows.

```
injective-authentication? Sender -> Recipient: variable
```

This property states that if (1) `Recipient` commits to executing the block directly after he receives `variable` and (2) `Sender`, `Recipient`, and all other principals in the same protocol instance are honest, then `Sender` executed the protocol up to and including sending `variable` *after* the `Recipient` started this protocol session.

Note that injective authentication does not capture all replay attacks. Specifically, the property does not mention that `Recipient` has to be the intended recipient of the message. Verifying those kinds of properties would require implementing support for additional queries.

We additionally support a special kind of sanity query, the `executable` query. This query states that all the principals are able to reach the end of the protocol state even if they are honest. All protocols that can be specified in our language are executable. An 'attack' to this query is often the correct execution of the protocol.

2.1.6 Additional declarations

We allow the user to specify some additional declarations about the protocol before the main body of a protocol specification. So far the only declarations supported are such that change the behavior of the compiler, i.e. the error messages that it prints out.

By default, mentioning a principal for the first time by specifying its first block will print an info message saying that the principal has not been declared. The principals of a protocol may be declared using a `principals:` declaration. For example, principals `Alice` and `Bob` are declared in example 9 by

```
principals: Alice, Bob
```

If a `principals:` declaration was made and an undeclared principal is mentioned then the compiler prints an error message and the compilation fails.

If an implicit assertion of equality occurs and there is not a matching unary `EQUALS` check then the compiler prints out an info message about it. The `unary-equals:` declaration followed by either `implicit` or `explicit` can change this behavior. Using

`implicit` will hide these info messages and using `explicit` will instead make the compiler print an error message and fail after encountering a missing unary equals check.

Using a declaration `hide-info: true` will make the compiler hide all info messages. Using a declaration `quit-on-warning: true` will make the compilation fail when a warning is encountered.

2.2 Steps to avoid user error

We perform sanity checks on security protocol specifications to warn users about most of the common mistakes. In this section, we describe these sanity checks and how they are performed.

Our main goal is to ensure that every protocol specified in our input language is executable. This is the case even if there is no interference from the adversary and if we disregard the possibility of unusual protocol instances. A protocol may not be executable (with regard to everything that is taken into account during the analysis) if it contains an assertion that can never succeed.

Example 12. Consider a protocol containing the following code

```
// warning: this specification is invalid
Principal [
  generates x, y
  EQUALS(x, y)?
]
```

This protocol is not executable because variables declared in `generates` statements are guaranteed to be distinct. The principal will never commit to executing this block because the `EQUALS` check never succeeds.

Because the standard Dolev-Yao adversary *is* the network, a protocol cannot succeed unless he decides to forward messages to the intended recipients. A protocol has to be executable even if the adversary acts just like a normal (but reliable) network.

Example 13. Consider a protocol containing the following code

```
// warning: this specification is invalid
Alice [
  generates x, y
]
Alice -> Bob: x, y
Bob [
  EQUALS(x, y)?
]
```

A protocol like this is executable according to the definition of the `executable?` query. The adversary can, for example, replace messages `x`, `y` with `x`, `x`. Both principals remain honest because no long-term reveal happened. Even though the protocol is executable, it clearly cannot represent any meaningful protocol because the adversary has to replace messages in transit to make it succeed.

The instantiation of public variables is less restrictive than that of private variables. Separate public variables can be instantiated to the same value or even to constants. In real-world protocols, this is very unusual or implausible. Protocols that rely on these instances are not meaningful. In most cases, they can be rewritten to protocols that do not violate this rule (by replacing multiple public variables with just one).

Example 14. Consider a protocol containing the following code

```
// warning: this specification is invalid
principals: Alice, Bob
Alice [
  knows public Bob
  EQUALS(Alice, Bob)?
]
```

This protocol is performed by two principals, `Alice` and `Bob`. Principal `Alice` only commits to executing this block if she is in fact communicating with herself. This is a valid instance of many real protocols, but it should never be the only instance.

2.2.1 Expected equality

We have described three cases when an equality assertion is performed on terms that are not expected to be equal. We generalize and formalize this notion of *expected equality* using an equational theory. We decide the equality of terms according to this theory efficiently using a convergent rewriting system. Most of the sanity checks performed by our compiler rely on expected equality.

The equational theory is specific to a protocol specification. For a protocol specification $\mathcal{P}$ we define an equational theory $=_{\mathcal{P}}$. We define an equation using $\simeq_{\mathcal{P}}$ to denote that it belongs to the set of equations generating $=_{\mathcal{P}}$.

To answer the first goal, variables declared in `distributed` statements or assignments must be equal to their corresponding term. Formally, if $\mathcal{P}$ defines a variable `v` in an assignment or a `distributed` statement and its corresponding term is `t` then

$$\texttt{v} \simeq_{\mathcal{P}} \texttt{t}$$

In section 2.1.2 we describe two equivalences that allow us to deconstruct encrypted

terms. These equivalences define two of the equations that generate $=_{\mathcal{P}}$:

$$\texttt{DEC(key, ENC(key, message))} \simeq_{\mathcal{P}} \texttt{message}$$

$$\texttt{ADEC(sk, AENC(PK(sk), message))} \simeq_{\mathcal{P}} \texttt{message}$$

We also describe equivalences that represent the properties of operation on cyclic groups. We cannot just include these equivalences because they use functions that do not exist in our specification. This will not cause any problems because our specification does not allow the user to deconstruct exponentiation. We only need equations that capture the associativity and commutativity of multiplication in exponents. Formally, if `b` (the base), `e1` and `e2` (the exponents) are any terms then

$$\texttt{b\^{}(e1\^{}e2)} \simeq_{\mathcal{P}} \texttt{b\^{}e1\^{}e2}$$

$$\texttt{b\^{}e1\^{}e2} \simeq_{\mathcal{P}} \texttt{b\^{}e2\^{}e1}$$

The equations so far define which terms have to be equal. The equations do not define any equality of variables that were defined in `knows` and `generates` statements. This means that the equations also capture the guarantee that values of private variables are distinct and the expectation that the values of public variables defined in `knows` are distinct.

Note that there are no equations that describe the properties of tuples as a simple concatenation of terms. For example, `{{x1, x2}, x3}` is the same as `{x1, x2, x3}` during the analysis. In protocol specification, we expect the user to use tuple structure consistently. This allows us to warn about mistakes like accidentally deconstructing the wrong tuple. A protocol can always be rewritten such that the structure of tuples is consistent.

Finally, expected equality needs to capture the behavior of a network that is reliable and is not controlled by an adversary. The message received has to be equal to the message sent. Formally, if there is a message statement

```
Sender -> Recipient : m
```

in $\mathcal{P}$ then

$$\texttt{Recipient's m)} \simeq_{\mathcal{P}} \texttt{Sender's m}$$

A message statement may send more than one message or even tuples. We kept the formal definition simple and consider all message statements broken into single-message statements.

2.2.2 Deciding expected equality

A straightforward way to decide equality of term with respect to $=_{\mathcal{P}}$ is by exhaustive search through all the possible variations of each term. A protocol is finite, so the

number of variations that a term is 'reasonably' equal to is finite as well. Reasonably because terms can always be enclosed in a decryption of an encryption, but these infinite variations can be simply avoided.

The time complexity of this approach is acceptable when deciding unification. Variables can be unified with anything, therefore it is not necessary to search through the entire protocol for variations of terms. Tamarin uses this approach to unify terms in its analysis [11, 14].

Deciding equality requires (and allows) a more efficient approach. The equations generating the equational theory $=_{\mathcal{P}}$ can be oriented to define a convergent rewriting system $\mathcal{R}_{\mathcal{P}}$. A convergent rewriting system defines a unique normal form of every term. Terms are equal with respect to $=_{\mathcal{P}}$ if they have the same normal form with respect to $\mathcal{R}_{\mathcal{P}}$.

Searching through all the variations of a term can have exponential time complexity with respect to the size of $\mathcal{P}$. Finding the normal form requires only a linear number of rewrites. Additionally, normal forms of previously encountered terms can be remembered. That way the number of rewrites needed to find the normal form of each new term is constant.

We construct $\mathcal{R}_{\mathcal{P}}$ by orienting all the equations generating $=_{\mathcal{P}}$, except for the equation `b^e1^e2` $\simeq_{\mathcal{P}}$ `b^e2^e1`. We orient the equation left-to-right as they were defined in the last section. For example, `v` $\simeq_{\mathcal{P}}$ `t` is oriented to `v` $\rightarrow_{\mathcal{P}}$ `t`. This rewriting relation rewrites terms such that either they are defined earlier in the protocol or they are simpler (shorter).

Orienting an equation that represents commutativity is more involved. If we just orient it in either direction, the resulting rewriting relation would not be convergent (would not be terminating). We need a deterministic way to decide which order of exponents, `b^e1^e2` or `b^e2^e1` is simpler. We use a method called ordered rewriting. We define a total ordering $\leq_{\mathcal{P}}$ on terms from $\mathcal{P}$. We define the rewriting system $\mathcal{R}_{\mathcal{P}}$ such that exponents rewrite to ascending order. Formally, if `e2` $\leq_{\mathcal{P}}$ `e1` then

`b^e1^e2` $\rightarrow_{\mathcal{P}}$ `b^e2^e1`

This increases the number of rewrites needed to find the normal form of a new exponentiation term to the complexity of a sorting algorithm. This complexity is still very reasonable.

Defining a suitable ordering is not straightforward. We have to avoid a situation where `a` $\lneq_{\mathcal{P}}$ `b` $\lneq_{\mathcal{P}}$ `c`, but `a` $=_{\mathcal{P}}$ `c`. Any ordering on the variables defined in `knows` and `generates` is valid because they are never expected to be equal. We can extend this ordering to all normal forms with respect to $\mathcal{R}_{\mathcal{P}}$ because they only contain these variables. The ordering can be extended to the remaining terms in $\mathcal{P}$ inductively. For

terms `l`, `r` we define that `l` $<_{\mathcal{P}}$ `r` if and only if $\mathcal{R}_{\mathcal{P}} \downarrow$ `l` $<_{\mathcal{P}}$ $\mathcal{R}_{\mathcal{P}} \downarrow$ `r`. Term $\mathcal{R}_{\mathcal{P}} \downarrow$ `t` denotes the normal form of `t` with respect to $\mathcal{R}_{\mathcal{P}}$.

Example 15. Consider the following part of a protocol

```
generates a, x
b = x
e = 'g'^b^a
```

The ordering $<_{\mathcal{P}}$ is defined on the variables from `generates` arbitrarily. Let us suppose it is lexicographical, `a` $<_{\mathcal{P}}$ `x`. The constant `'g'` is in normal form, we arbitrarily extend the ordering to it. Let us suppose we use `'g'` $<_{\mathcal{P}}$ `a` $<_{\mathcal{P}}$ `x`. Variable `x` is the normal form of `b` , therefore `a` $<_{\mathcal{P}}$ `b`. This means that `'g'^b^a` rewrites to `'g'^a^b` and further to the normal form `'g'^a^x`.

2.2.3 Utilizing expected equality

In this section, we note all the verifications performed by our compiler that utilize expected equality. For the remainder of this section, when we state that terms have to be equal, we mean equality with respect to $=_{\mathcal{P}}$. In other words, they have to be expected to be equal. Expected equality applies in the following scenarios:

- a binary `EQUALS` check is used
  - the arguments of this check have to be equal
- an implicit equality assertion occurs (it does not matter if a matching unary `EQUALS` check follows)
  - the compared variables have to be equal
- a `SIGNVERIF` check is used
  - the first argument has to be equal to a public key
  - the last argument has to be equal to a signature
  - the secret key in the public key has to be equal to the secret key in the signature
  - the middle argument has to be equal to the message in the signature
- a tuple is used on the left side of an assignment
  - the right side of the assignment has to be equal to a tuple with the same structure
- symmetric decryption is used
  - the second argument has to be equal to a symmetric encryption
  - the first argument has to be equal to the key in the symmetric encryption
- asymmetric encryption is used
  - the first argument has to be equal to a public key
- asymmetric decryption is used
  - the second argument has to be equal to an asymmetric encryption

  - the first argument has to be equal to the secret key in the asymmetric encryption

2.2.4 Remaining sanity checks

There are other sanity checks performed by the compiler that do not rely on expected equality. Most of the remaining properties of a valid protocol are stated in this section.

- principal identifiers cannot cause collisions with variable identifiers
- principals should be declared (no declaration triggers info messages, a declaration with a missing principal triggers an error)
- every principal has to execute some statements within a protocol session (not only initial assumptions)
- variable identifiers should not cause collisions (in some cases, variables may be shadowed but this triggers a warning message)
- every variable used has to be declared and it has to be known to the principal that is attempting to use it
- the construction within a `distributed` statement can only contain long-term public variables
- implicit equality assertions should have matching unary `EQUALS` checks
- the parameters of confidentiality and forward secrecy queries have to be valid variables
- the messages in authentication and injective authentication queries have to be defined in the protocol
- functions must have the correct number of arguments
- messages and left sides of assignments can only contain constants, variables, and tuples

2.3 Translating our language to Tamarin

In this section, we describe the process used to translate our input language described in section 2.1 into Tamarin input language described in section 1.3.2.

2.3.1 Protocol structure

When translating a protocol specification to Tamarin, we need to capture some properties, for which Tamarin does not have an explicit representation. A statement in our language has an explicitly defined principal (variable) that performs it. The principal performing a statement is defined by the principal block or message statement. Most statements in our language represent sequential actions. This means that the state-

ments preceding them have to be finished before they can be performed. Even the statements that represent declarations about the initial assumptions have a sequential meaning. They have to be finished before a protocol session can be initiated.

The protocol specification in Tamarin is expressed using rules. The properties we mentioned can be captured using custom facts. Specifically, a statement `S` is translated to a Tamarin rule `rule S: [p]--[a]->[c]` where `p`, `a`, and `c` are comma-separated lists of premises, actions, and conclusions respectively. A principal `A` performing `S` defines a role A that represents one party of a protocol. This role can perform `S` if he has reached state p directly before `S`. By performing `S`, he will reach state q (such that $p \neq q$) directly after `S`. These properties can be captured by linear facts `A_p()` and `A_q()`. Fact `A_p()` is established if there is a principal in the role of A and he has reached the state p. Fact `A_q()` is established by performing the statement `S`. The Tamarin rule representing `S` can then be extended to `rule S: [p, A_p()]--[a]->[c, A_q()]`.

These facts still only capture the existence of a role A and whether any principal in this role reached certain points in some undefined session of the protocol. There is no way to distinguish between the various principals acting in the role of A (instantiations of `A`). If several protocol sessions are initiated then there is no way to distinguish which one has reached the state p. If the initial assumptions of the protocol are instantiated with different long-term values then there is no way to distinguish between these instances. To solve these three problems, the facts additionally contain variables specifying these three properties of a state. The fact `A_p(i,j,A)` denotes that principal `A` in the role of A has reached state p while executing session `j` with the initial assumptions given by protocol instance `i`. Variable `A` is instantiated to a public name that is bound to role A when instantiating the initial assumptions. Variable `j` is instantiated to a fresh name unique to a protocol instance. Variable `i` is instantiated to a fresh name unique to a protocol session. These variables are simply carried over to the `A_q` fact, so the rule is extended as follows.

```
rule S: [p, A_p(i, j, A)]--[a]->[c, A_q(i, j, A)]
```

All statements either add to the knowledge of the principal or require the principal to know some variables. For example, an assignment requires that the variables on its right side are known and adds new variables from the left side to the knowledge of the principal. Principal knowledge also needs to be tracked in Tamarin so that variables created by rules can be used later. We track knowledge variables in the state facts as well. Therefore, if the principal knows variables `v1`, ..., `vk` in state p and learns `u1`, ..., `ul` by executing statement `S` then we extend the rule representing this statement in Tamarin as follows.

```
rule S: [
```

```
  p,
  A_p(i, j, A, v1, ..., vk)
  ]--[a]->[
  c,
  A_q(i, j, A, v1, ..., vk, u1, ..., ul)
]
```

Some variables need to be prefixed by `~` or `$` to indicate that they can only be instantiated with fresh or public names. This would make a proper definition too complicated. Instead, we provide an example. The way sorts are assigned to variables is described in 2.3.2.

Example 16. Consider the following protocol specification.

```
Alice [
  knows public K
  generates N
]
```

Let us define the states before and after the statement `generates N` as p and q respectively. This statement is translated using a premise `Fr(~N)`. The Tamarin rule representing this statement could be the following.

```
rule Gen: [
  Fr(~N),
  Alice_p(~i, ~j, $A, $K)
] --[] -> [
  Alice_q(~i, ~j, $A, $K, ~N)
]
```

Using the procedure defined so far, we could translate a protocol such that each statement corresponds to one rule. All statements would occur in exact sequential order. Doing this would result in extensive protocol specifications in Tamarin. In many cases, the order of execution of certain statements does not have any effect on security in our model.

The only actions that produce observable side effects are sending messages and finishing the protocol. This means that the sequential order of many of the statements is irrelevant. We have already described what it means when a principal commits to executing a block in section 2.1.5. We combine statements such that each rule in Tamarin represents a principal committing to executing a block.

If there are two blocks performed by the same principal with no messages sent by this principal between them, these blocks are merged. Each rule also contains all messages received by the principal between this and the previous (merged) block as well as all messages sent by the principal between this and the next block. Each rule contains only one premise state fact and one conclusion state fact.

The statements declaring initial assumptions of a protocol are not translated in their corresponding block. Instead, they are all combined to create a single rule that initiates the protocol instance. The initial states created in an instance are persistent. This allows principals to run an unbounded number of sessions in the same instance. A protocol session is initiated by another rule that only adds the session variable (`i` in our examples) to the initial states of principals. The initial states created in a session are already linear (non-persistent).

Example 17. Recall example 9. All the initial assumptions are in separate blocks. After they are removed and combined into the instance rule, their blocks will be empty. Principals do not send any messages between their respective initial assumption block and their next one. This means that the empty blocks get merged. The resulting Tamarin code will contain one rule for each principal.

2.3.2 Constants, variables, and cryptographic functions

The translation of messages (terms) is very straightforward. All of the constructs that we use to build messages also exist in Tamarin, only with different syntax.

Constants, the simplest, are copied directly. For example, a constant `'constant'` in our input language is translated to the public name `'constant'` in Tamarin.

Translating variables is more involved. We gather variables based on their identifiers and the scope in which they are used. We assign a global sequential number to every variable. We translate them to Tamarin identifiers as the sequential number with a predefined prefix. Tamarin keeps variable identifiers when instantiating them to names. Sometimes Tamarin needs to use new variables and new names. We want to avoid collisions with these names so that we can identify variables from our input in the output provided by Tamarin. We have observed Tamarin using prefixes `t`, `x`, and `c` when naming new variables. We cannot be sure if there are other prefixes that Tamarin uses or might use in future versions, but it seems to prefer single-character prefixes. We chose prefix `var` for our naming convention. Variables may additionally be prefixed with a sort symbol, `~` or `$`. For example, a generated ephemeral variable in our input language that is assigned the sequential number 20 will be translated to `~var20`.

Variables in Tamarin may be prefixed with a sort symbol, `~` or `$`. This dictates the sort of the names that can be instantiated for the variables. Sometimes the addition of sort symbols is mandatory. All variables used in a `Fr` fact have to be marked as fresh (`~`). All variables that are not mentioned in the premises because they should be instantiated as public, have to be marked as public `$`. Otherwise, the addition of sorts is optional because it may incur additional limitations on the protocol that change the security properties. For example, if an `In` fact contains a fresh variable, it means that

the principal does not accept a public value in this message. This may be impossible to verify unless the principal knows the exact value that is supposed to be in that message. When the addition of sort symbols does not change the security properties, we want to provide them for Tamarin. Tamarin evaluates all the equal variations of each term when loading a protocol. This may cause extreme loading times when operations on cyclic groups are used. By adding sort symbols, we limit the possible variations. This allows the protocol to load in some cases where it otherwise would not. We dedicate the case study 3.4 to this topic. We add the optional sort symbol to all the variables that the principal knows (directly, without regard to equality).

Cryptographic functions are translated using Tamarin's built-in theories. They define functions and equations generating the equational theory $=_{\mathcal{E}}$, described in 1.3.1. We denote that the set of equations generating $=_{\mathcal{E}}$ contains an equation by defining it with $\simeq_{\mathcal{E}}$. We have adopted syntax from VerifPal [9]. VerifPal and Tamarin syntax use different, but self-explanatory, function symbols. The order of arguments is often reversed in Tamarin.

We use theory `symmetric-encryption` to translate functions `ENC` and `DEC`. This theory defines the function symbols `enc` and `dec`, and an equation

`dec(enc(message, key) key)` $\simeq_{\mathcal{E}}$ `message`

We use theories `asymmetric-encryption` and `signing` to translate functions `PK`, `AENC`, `ADEC`, and `SIGN`. These theories define the function symbols `pk`, `aenc`, `adec`, `sign`, `verify`, and `true` (`true` is a constant function), and equations

`adec(enc(message, pk(sk), sk)` $\simeq_{\mathcal{E}}$ `message`

`verify(sign(message, sk), message, pk(sk))` $\simeq_{\mathcal{E}}$ `true`

We use theory `hashing` that defines the function symbol `h` to translate `HASH`. There are no equations associated with hashing.

We use theory `diffie-hellman` to translate exponentiation (`^`) in public groups. This theory defines function symbols `^` (exponentiation), `*` (multiplication), `inv` (inverse), and `1` (a constant function 1). The theory defines the properties of cyclic groups using these equations:

`x ^ y ^ z` $\simeq_{\mathcal{E}}$ `x ^ y * z` `x ^ 1` $\simeq_{\mathcal{E}}$ `x` `x * y` $\simeq_{\mathcal{E}}$ `y * x`

`(x * y) * z` $\simeq_{\mathcal{E}}$ `(x * y) * z` `x * 1` $\simeq_{\mathcal{E}}$ `x` `x * inv(x)` $\simeq_{\mathcal{E}}$ `1`

Tamarin expresses operations on tuples using a default theory that contains a symbol for a pair and deconstruction symbols for retrieving the first or second element of a pair. Using these symbols is inconvenient. We use a special syntax supported by Tamarin to translate tuples. A tuple `{t1,...tk}` is translated to `<t1, ..., tk>`, which tamarin breaks up into pairs during compilation. The properties of tuples remain as we defined them.

2.3.3 Assignments

The most complicated statement to translate is an assignment. In cases of assignments that construct terms, we can simply translate them to an alias before their corresponding rule.

Example 18. The assignment

```
c = ENC(k, m)
```

may be translated to an alias

```
var1 = enc(var2, var3)
```

if `c`, `k`, `m` are translated to `var1`, `var3`, `var2` respectively.

We have to translate tuple deconstructions differently. We translate other deconstruction the same way as tuple deconstructions to remain consistent. In Tamarin, tuples cannot be used on the left sides of aliases. On the other hand, any terms can be used almost anywhere else. When an assignment contains a deconstruction, then we do not add any new parts to the corresponding rule, but remember that it happened in the corresponding block. We then replace the deconstructed variable with a term that shows how it was constructed. We only replace the variable in the rule corresponding to the deconstruction.

Example 19. Suppose there is an assignment

```
{x1, {x2, x3}} = t
```

Then within the rule that contains it, we replace the translation of `t` with the translation of `{x1, {x2, x3}}`. It will be added to the resulting state of the rule as `{x1, {x2, x3}}`, but the next rule can retrieve it as `t` and use it normally.

This might accidentally create additional assumptions that would change the security of the protocol. For example, when we use the deconstructed tuple inside an `In` fact, we assume that the message received must be a tuple. The sanity checks performed by our compiler protect us from this problem. A term cannot be deconstructed unless it is expected to have the correct structure. This means we do not make any *new* assumptions about the structure of terms.

Example 20. Suppose that a message is encrypted as

```
c = ENC(k, m)
```

and a principal decrypts it as

```
n = DEC(k, c)
```

The translation of `c` will be replaced with the translation of `ENC(k, n)` within the rule. If `c` was received in a message, then the principal will only accept messages with

the same structure as `ENC(k, n)`. The principal also has to know `k`. Both of these things we verified in sanity checks.

2.3.4 Statements

Translating the other statements is much simpler. Each statement just adds some parts to the corresponding rule or the initial assumptions rule.

The `knows public` statement adds the public variables to the principal states created in the initial assumptions rule. The `knows private` statement adds `Fr` facts for the private variables to the premise of the initial assumptions rule and also adds the variables to principal states.

The `distributed` statement adds an alias for the construction of the public variable to the initial assumptions rule. We say the variable is public because that is the representation. It is not directly instantiated, it is set equal to the term that constructs it. This means it does not have the public sort and does not get prefixed by `$`. We also add the variable to the state of the principal that declared it.

The `generates` statement adds `Fr` facts for the ephemeral variables to the premise of the corresponding rule and also adds the variables to the resulting state of the rule.

Message statements add `Out` facts to the conclusions of the sender's previous rule and `In` facts to the premise of the recipient's next rule. They also add the received variables to the recipient's state.

Checks are translated using the equality restriction described in section 1.3.2. The restriction states that the arguments of an `Eq` fact must be equal with respect to $=_{\mathcal{E}}$. The `EQUALS` check just adds an `Eq` fact with the translated arguments to the actions of the corresponding rule. The `SIGNVERIF` translates the arguments into a `verify` term and puts it equal to `true` using an `Eq` fact.

2.3.5 Queries

We represent long-term reveals by a custom rule for each principal. The rule has the principal's initial state in premises. It marks the principal as dishonest by a `dishonest` fact in actions. It sends all long-term private variables of the principal to the adversary using `Out` facts in conclusions.

Lemmas can only reason about facts that occur in actions of rules. It is often convenient to reason about the resulting states of rules because they contain all the necessary variables. For this reason, we add exact copies of the resulting states from conclusions of rules to their actions. When we defined queries in 2.1.5, we often mentioned that principals commit to executing a block. This means that the result state action fact of the corresponding rule is established.

In order to have a list of all principals bound to a protocol instance, we add a `principals` fact to the actions of the initial assumptions rule. This allows us to translate assumptions about all the principals in the same instance as the principal from the query. We use the variable that identifies the instance as the first argument and the identity variables of principals as the remaining arguments.

The fact that the adversary learns a variable is expressed by a `K` action fact. This fact can be established by Tamarin's predefined rules described in 1.3.1.

Specifically for each authentication (injective and non-injective) query, we add a custom action fact to the rule where the sender sends the variable. The use of the result state fact in this case would incur more assumptions than we want to. These custom facts only contain the sender's identity variable and the variable that was sent.

Any formal templates for the individual queries are too long and complicated to be stated here due to quantification. At the same time, the translation of queries reflects the descriptions in section 2.1.5. We only reword the formulas so that they use less negation. In theory, Tamarin should only accept guarded formulas ([11, 14]). Improvements to Tamarin have loosened this requirement. Reducing the use of negations is sufficient to make Tamarin accept our lemmas.

Example 21. The first confidentiality query from 9 is translated as

```
lemma secrecy_0:
  all-traces "
    All var0 var2 var3 #t0 var1 var5 var6 var7 var8 var9 var10 #t1 #t2.
        Principals(var0, var2, var3) @ #t0 &
        Pal3_1(var0, var1, var3, var5, var6,
                           var7, var8, var9, var10) @ #t1 &
        K(var7) @ #t2
      ==>
        (Ex #t3.Dishonest(var2) @ #t3) |
        (Ex #t4.Dishonest(var3) @ #t4)
  "
```

The state fact is named `Pal3_1`. The `Pal3` represents that it belongs to the principal with identity variable `var3`, this is `Bob`. The `1` is the sequential number of this block within the entire protocol (including Alice's blocks). The lemma should be interpreted as follows. If the instance is `var0`, principals are `var2` and `var3`, principal `var3` executes block `1` (this is the first one where he knows `var7`), and the adversary knows `var7`, then one of the principals must be dishonest.

2.4 Translating Tamarin-Prover output

Tamarin-Prover provides output from its analysis in multiple forms. It is mainly intended to be used within an interactive graphical interface. This interface is provided

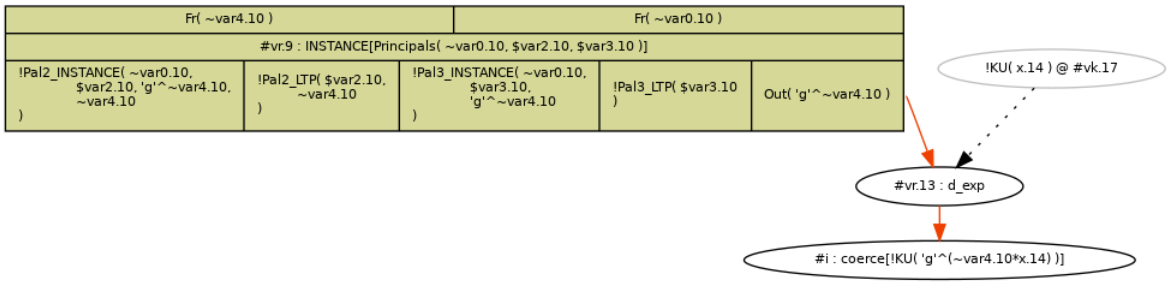

Figure 2.1: Example of a source

as a local website. It displays the traces of attacks as rendered images of graphs. During any computation, Tamarin prints logging messages to the error output of the terminal. After the analysis of a lemma is finished, it also prints the steps needed to reconstruct an attack or proof of security.

We decided to attempt to translate the text output provided by Tamarin in the terminal and disregard the interactive interface. We believed that translating the interactive interface would be too time-consuming to be implemented within the scope of this thesis. This decision proved to be incorrect because the text output of Tamarin-Prover is often insufficient to manually reconstruct an attack. We attempted to overcome this by extending the output provided by Tamarin-Prover. This allowed us to present results that are more complete, but still cannot be considered simple or intuitive.

We understand that translating the interactive interface likely would have been more successful. It would not require alterations to Tamarin-Prover and the graphical presentation would be more intuitive.

2.4.1 Extending the output of Tamarin-Prover

Tamarin-Prover creates precomputed sources for facts that appear in a protocol. These sources are created during loading times. They describe steps that need to be taken in order to establish a fact.

Example 22. Figure 2.1 shows a graph of a simple source. This is one of the sources that show how the adversary may learn a term that has the same structure as `t.1^t.2`. This source specifically allows the adversary to learn a term that has the same structure as `'g'^(~var4.10*x.14)`. The oval in the center states that he attempts to construct it by exponentiation. The square on the left shows a protocol rule. In this case, it is a rule called `INSTANCE`. We use this rule to define the initial assumptions of a protocol. The adversary uses this rule to learn a term `'g'^~var4.10`. The rule sends out this term because it represents a public key and as such the adversary should know it. On the right side, the rule shows that the adversary also needs to learn `x.14`, but this source does not describe how.

The other forms of output reference these sources. They state the fact that is being solved and the name of the source used to solve it. This means that in order to translate the text output of Tamarin-Prover, we need access to a text representation of the sources. The sources are exported for a graph rendering tool while the interactive interface is running. We wanted to avoid running a web server because our application only provides output in form of text. We made a minor change to Tamarin-Prover that prints the complete graphs for sources when running without the interactive interface. These graphs are provided in JSON format. They are extensively long even for small sources, so we do not provide an example.

The logging messages printed during computation contain the solved goal (the established fact), but not the source used to solve it. We made another change to Tamarin-Prover that also prints the name of the source. Tamarin-Prover performs multiple computations in parallel. As a result, the solved goals and sources may sometimes change order. We also added an index number to the goal and the source so that they can be paired when such a change in their order happens.

2.4.2 Translating the extended output

We translate source graphs into a text file called `sources.txt`. Some sources do not provide interesting information. These include sources of custom facts that represent a principal committing to executing a block or internal facts like long-term reveals. Their sources only state that they must originate from their respective rules. We omit these sources to make the output more concise. The sources file only contains sources that describe how the adversary learns terms with a specific structure.

Sources are grouped if they provide the same fact (allow the adversary to learn terms with the same structure). Some groups are more specific than others. This is defined by matching on the terms that they provide to the adversary. Term s matches term t if there is a substitution σ such that $s =_{\mathcal{E}} t\sigma$. If s matches t then s is at least as specific as t. For example, `t1^t2` matches `t1` because we can substitute the entire term `t1^t2` for the variable `t1`. This does not work the other way, therefore `t1^t2` is strictly more specific than `t1`. We also say that the group providing `t1^t2` is more specific than the group providing `t1`. More specific groups are more likely to be used in a valid attack. We order groups from the more specific to the less specific.

Sources contain too much information to be comprehendible in text format. We try to extract the most relevant information and omit the rest.

Example 23. Translating all the information in the rule used in 2.1 would create a very convoluted output. The important piece of information is that `'g'^~var4.10` was received from the initial assumptions. The variable `~var4` is the variable `as` in the input and we translate it back. We translate the entire source as follows.

```
solve: adversary learns t.1^t.2
Source (7) initial assumptions
Adversary coerces the protocol to use 'g'^as.10*x.14
  Adversary deconstructs 'g'^as.10*x.14 using ^ on messages from:
    Adversary receives 'g'^as.10 in a protocol message
      Adversary learns it as a public value from initial assumptions.
    Adversary gets x.14 from another source.
```

The rest of the output consists of the logging messages and the resulting attack or proof of security. These have a similar format after our extensions. They consist of pairs – goal and source name. The goal contains the term that was learned by the adversary and the source names identify the source that was used for it. Translating these is slightly ambiguous because the names of sources are not unique. The output is not ambiguous when interpreted by Tamarin-Prover and displayed in the interactive interface. Tamarin-Prover uses the same heuristics as the ones used in the analysis to assign the correct sources to their names. Interpreting the source names precisely would require reconstructing the analysis of Tamarin-Prover, which is entirely unreasonable. Instead, we list all the sources that are applicable to the given goal. We find all the source groups that the goal matches. More precisely, we find groups where the term learned by the adversary is at most as specific as the term in the goal. For example, the group that the source in 2.1 belongs to is applicable to a goal with the term `'g'^a*b` because `'g'^a*b` matches `t1^t2`. From these applicable groups, we list the sources that have the correct name. In most cases, there is only one applicable source. In cases when there are more applicable sources, they are provided ordered based on their groups, so the correct source is often first.

There are other types of logging messages that we did not yet manage to translate. We skip these and print a message that some messages could not be parsed. This mostly occurs in complex protocols.

Example 24. The second query from example 9 produces the following output.

```
property: confidentiality? Alice's m
DISPROVED
adversary learns ENC(bp^as, m)
applicable: 2
  adversary learns bp^as
  applicable: 6, 7
    success
```

When we inspect the source file we find out that source 2 states that adversary constructs `ENC(bp^as, m)` from `bp^as` and `m`. Source number 6 does not seem to let the adversary construct `bp^as`. Source number 7 is the source from 2.1. We can see that `bp^as` can be interpreted as `'g'^x*as` and the adversary can learn `'g'^as` from the

initial assumptions. To complete the attack, the adversary has to generate `x` and `m`. We cannot learn this last step from the available output.

Chapter 3

Case studies

This chapter aims to demonstrate the correctness of our implementation on several sample protocols. Our program is best suited for the analysis of short and simple protocols. We show that it is also capable of analyzing some complex, modern protocols by analyzing a Noise protocol [13] in section 3.3. A separate case study in section 3.4 demonstrates a procedure that improves the loading times of Tamarin-Prover.

The Diffie-Hellman exchange was thoroughly analyzed in example 9. The complete protocol specification can be found along with other case studies in the files associated with this thesis. These files also contain our implementation and installation instructions.

3.1 The Otway Rees protocol

The Otway Rees protocol [12] is susceptible to an attack that makes use of a type flaw [5]. This attack does not require the adversary to compromise any variables. This makes it ideal to demonstrate our weak confidentiality query.

We use a declaration to hide the info messages about missing unary `EQUALS` checks. This allows us to omit the unary `EQUALS` checks, but the implicit equality assertions still occur.

The protocol establishes a symmetric key between two principals. The key is provided by a trusted server. The initial assumptions state that `Server` shares a symmetric key `Kas` with `Alice` and a symmetric key `Kbs` with `Bob`. Principal `Alice` also needs to know the identity of `Bob` to inform the `Server` about the intended recipient.

```
principals: Alice, Bob, Server
unary-equals: implicit
Alice [
  knows private Kas
  knows public Bob
]
```

```
Bob [
  knows private Kbs
]
Server [
  knows private Kas, Kbs
]
```

Principal `Alice` initiates the protocol. She generates a nonce `N` that identifies the session and her own nonce `Na`. She sends an initial message to `Bob` as follows.

```
Alice [
  generates N, Na
  m1 = ENC(Kas, {Na, N, Alice, Bob})
]
Alice -> Bob: N, Alice, Bob, m1
```

Principal `Bob` generates his own nonce `Nb`. He creates a message with the same format. He forwards both messages to the `Server`. Note that `Bob` does not know the identity of `Alice`. He will use any identity received in the initial message. The attack does not misuse this behavior. We only point it out to demonstrate these semantics.

```
Bob [
  generates Nb
  m2 = ENC(Kbs, {Nb, N, Alice, Bob})
]
Bob -> Server: N, Alice, Bob, m1, m2
```

The server deconstructs both messages, verifies that all values are correct, and encrypts a message with a newly generated key `Kab` for each principal.

```
Server [
  {Na, N, Alice, Bob} = DEC(Kas, m1)
  {Nb, N, Alice, Bob} = DEC(Kbs, m2)

  generates Kab
  m3 = ENC(Kbs, {Nb, Kab})
  m4 = ENC(Kas, {Na, Kab})
]
```

The `Server` sends both messages to `Bob` and `Bob` forwards the second message to `Alice`. Both `Bob` and `Alice` retrieve `Kab` from the message encrypted by the key that they know.

```
Server -> Bob: N, m3, m4

Bob [
  {Nb, Kab} = DEC(Kbs, m3)
]
Bob -> Alice: N, m4
```

```
  Alice [
    {Na, Kab} = DEC(Kas, m4)
  ]
```

The protocol should guarantee that `Kab` is only known to `Alice`, `Bob`, and `Server`. The adversary may misuse the fact that the initial messages sent by `Alice` and `Bob` have a similar structure as the messages containing `Kab`. The adversary may replay the initial messages to the principals. This makes them accept the triple `{N, Alice, Bob}` instead of `Kab`. The triple `{N, Alice, Bob}` only contains public variables. This means that the symmetric key established by the protocol will be known to the adversary. The actual key `Kab` produced by `Server` remains confidential.

Specifically, `Alice` receives `ENC(Kas, {Na, N, Alice, Bob})` as message `m4` instead of `ENC(Kas, {Na, Kab})`. Both messages are encoded with `Kas`. Both messages contain a tuple beginning with `Na`. The only difference is that the original message ends with `Kab` and the fake message ends with `{N, Alice, Bob}`. Against `Bob`, the attack is very similar.

We specify queries to verify the confidentiality of all 3 different values of `Kab`. We expect to discover the described attack against `Alice's Kab` and `Bob's Kab`. The `Server's Kab` should remain confidential.

```
queries [
  confidentiality? Alice's Kab
  confidentiality? Bob's Kab
  confidentiality? Server's Kab
]
```

The first two queries find the expected attacks. The analysis of the third query does not terminate. Nontermination is sometimes unavoidable because Tamarin attempts to solve an undecidable problem. This is more common in cases where the analyzed property holds because the analysis has to check all possible executions instead of finding just one counterexample. The files associated with this thesis contain some protocols where the analysis does not terminate even though they are susceptible to an attack.

The attack against the first query contains the following goal-source pair.

```
  adversary learns ENC(Kas, {Na, Kab})
  applicable: 5, 17, 18, 19
```

This is the goal we are most interested in because it shows us, how message `m4` was constructed in the attack. The first applicable source (number 5) is the one we were looking for.

```
solve: adversary learns ENC(t.2, t.1)
Source (5) Alice's block nr. 2
Adversary coerces the protocol to use
```

```
ENC(Kas.10, {Na.10, N.10, Alice.17, Bob.10})
  Adversary receives ENC(Kas.10, {Na.10, N.10, Alice.17, Bob.10})
  in a protocol message
    which he receives after Alice's block nr. 2
      Protocol generates N.10
      Protocol generates Na.10
      Initial assumptions precede it.
```

Right from the name of the source we see that the target term, `ENC(Kas, {Na, Kab})` was reached using `Alice's block nr. 2`. Looking back at the protocol, this is the block after which `Alice` sends `m1`. We see that the adversary coerced the protocol to use some instance of `ENC(Kas, {Na, N, Alice, Bob}` instead of the target term `ENC(Kas, {Na, Kab})`. He received it from a message after `Alice's block nr. 2`. The rest of the source describes some parts of the protocol that we already know from the specification.

The second query produces the following goal-source pair.

```
  adversary learns ENC(Kbs, {Nb, Kab})
  applicable: 6, 20
```

Once again the first applicable source (number 6) is the correct one.

```
solve: adversary learns ENC(t.2, t.1)
Source (6) Bob's block nr. 2
Adversary coerces the protocol to use
ENC(Kbs.10, {Nb.10, N.10, Alice.10, Bob.24})
  Adversary receives
  ENC(Kbs.10, {Nb.10, N.10, Alice.10, Bob.24}) in a protocol message
    which he receives after Bob's block nr. 2
      Adversary sends N.10
        Adversary gets N.10 from another source.
      Adversary sends Alice.10
        Adversary gets Alice.10 from another source.
      Adversary sends Bob.24
        Adversary gets Bob.24 from another source.
      Adversary sends m1.10
        Adversary gets m1.10 from another source.
      Protocol generates Nb.10
      Initial assumptions precede it.
```

In this case, the adversary needs to send `m3` to the protocol. It must have the same structure as `ENC(Kbs, {Nb, Kab})`. Instead, the adversary replays some instance of `m2` that he receives from `Bob's block nr. 2`. The adversary has to send all the necessary messages to `Bob` in order to initiate his second block.

3.2 The Yahalom protocol

In this case study, we examine the Yahalom protocol featured in [4]. Our goal is to find an attack that abuses a type flaw described in [5]. We use naming conventions similar to the ones used in our case study of the Otway Rees protocol in section 3.1. The protocol aims to establish a shared key between two principals, `Alice` and `Bob`. The key is provided by a trusted server, `Server`. Each of the principals, `Alice` and `Bob`, shares a symmetric key with `Server`.

```
principals: Alice, Bob, Server
unary-equals: implicit
Alice [
  knows private Kas
]
Bob [
  knows private Kbs
]
Server [
  knows private Kas, Kbs
]
```

The protocol is initiated by `Alice`. She generates a nonce `Na` and sends it to `Bob` along with her identity. Then `Bob` generates his own nonce and requests a key from `Server` using the following messages.

```
Alice [
generates Na
]
Alice -> Bob: Alice, Na

Bob [
  generates Nb
  m1 = ENC(Kbs, {Alice, Na, Nb})
]
Bob -> Server: Bob, m1
```

The `Server` generates a key `Kab` and encrypts two messages, `ma` intended for `Alice` and `mb` intended for `Bob`. It sends both messages to `Alice` and she will later forward `mb` to `Bob`.

```
Server [
  {Alice, Na, Nb} = DEC(Kbs, m1)
  generates Kab
  ma = ENC(Kas, {Kab, Na, Nb})
  mb = ENC(Kbs, {Alice, Kab})
]
Server -> Alice: ma, mb
```

Before `Alice` forwards `mb` to `Bob`, she recovers `Kab` and uses it to encrypt the nonce `Nb`. This message is supposed to authenticate `Alice` to `Bob` as someone who knows `Kab` and subsequently also `Kas`. For that reason, we call this encrypted message `auth`.

```
Alice [
  {Kab, Na, Nb} = DEC(Kas, ma)
  auth = ENC(Kab, Nb)
]
Alice -> Bob: mb, auth

Bob [
  {Alice, Kab} = DEC(Kbs, mb)
  Nb = DEC(Kab, auth)
]
```

The protocol should guarantee the confidentiality of `Kab` for each of the principals and `Alice` should be able to authenticate to `Bob` using message `auth`. Unfortunately, the analysis of these properties does not terminate. Non-termination provides a weak indication that the properties are not susceptible to attacks.

The attack cannot be employed against the protocol as it is. It requires the adversary to learn `Nb` during the session. He can replace message `mb` with the initial message `m1`. More precisely, the message `ENC(Kbs, {Alice, Kab})` is replaced with `ENC(Kbs, {Alice, Na, Nb})`. This makes `Bob` accept the pair `{Na, Nb}` instead of a key `Kab` generated by `Server`. The adversary needs to know `Nb` because he needs to construct the authentication message `auth` as `ENC({Na, Nb}, Nb)`. We publish `Nb` by sending it to `Server` along with the initial message.

```
Bob -> Server: Bob, m1, Nb
```

We can find the attack using either of the following queries.

```
authentication? Alice -> Bob: auth
confidentiality? Bob's Kab
```

The first query aims to find out that the authentication message `auth` can be constructed by the adversary. The second query should find out that `Bob` can be forced to accept a key known to the adversary. The results of these queries are identical and contain the following lines.

```
adversary learns ENC(Kbs, {Alice, Kab})
applicable: 8, 23, 24
  adversary learns ENC({Na.1, Nb.1}, Nb)
  applicable: 5
```

The first pair shows how the adversary learned message `mb`. We know that this message should have been replaced by `m1`. The first applicable source (number 8) shows the main idea behind this attack.

```
solve: adversary learns ENC(t.2, t.1)
Source (8) Bob's block nr. 2
Adversary coerces the protocol to use
ENC(Kbs.10, {Alice.10, Na.10, Nb.10})
  Adversary receives
  ENC(Kbs.10, {Alice.10, Na.10, Nb.10}) in a protocol message
    which he receives after Bob's block nr. 2
      Adversary sends Alice.10
        Adversary gets Alice.10 from another source.
      Adversary sends Na.10
        Adversary gets Na.10 from another source.
      Protocol generates Nb.10
      Initial assumptions precede it.
```

We can see that the adversary coerced the protocol to use some instance of the replayed message `ENC(Kbs, {Alice, Na, Nb})`. It also shows that he received this message after `Bob's block nr. 2`. This is the block when `Bob` sent `m1`. The rest of the source states the requirements for initiating this block.

The second pair shows us that the adversary also needed `ENC({Na, Nb}, Nb)`. We know this to be the replacement for the message `auth`. Tamarin found the attack using two different sessions and had to distinguish `Nb` and `Nb.1`. This was not necessary but it is equally correct.

3.3 Noise protocol

This case study aims to demonstrate that our program can be used to analyze some complex, modern protocols and produces the expected results. We chose the "IK" pattern from The Noise Protocol Framework for this demonstration.

Noise is a framework for cryptographic protocols based on the Diffie-Hellman key agreement. Its main purpose is to describe handshakes that establish a symmetric key between two parties. The complete technical description of the Noise framework is outside of the scope of this case study. We only explain the features that are captured by our model. The computational details are described in [13].

3.3.1 Cryptographic functions

The framework specification denotes a Diffie-Hellman keypair as a single variable (object). It also uses keypairs to specify inputs for Diffie-Hellman calculations. We specify Diffie-Hellman calculations explicitly as exponentiations, which requires addressing the secret and public key separately. The framework also addresses the parties of the protocol as the initiator and the responder or the local and the remote (in addition to Alice and Bob). We only address the parties as Alice and Bob to avoid any confusion.

This allows us to use a consistent 3-character notation for keys. The first character is `S` for static or `E` for ephemeral. The second is `S` for secret or `P` for public. The third is `A` for Alice or `B` for Bob. For example, the static public key of Alice is denoted as `SPA`. The DH computation between the ephemeral keypair of Bob and the static keypair of Alice is performed as `SPA^ESB` by Bob and as `EPB^SSA` by Alice.

Encryption is performed as authenticated encryption with associated data (AEAD). This produces ciphertext and authentication data that are sent together as a single message. We can produce the ciphertext using function `ENC` and the authentication data using function `HASH` (representing HMAC in this case). After decryption, the recipient reproduces the same authentication data and checks that it matches the received. AEAD encryption also takes a nonce as an argument, besides the key, the plaintext, and the associated data. This nonce is produced as a simple counter, which makes it constant for a selected protocol. It does not have any meaningful effect on the protocol in our model. We omit it to reduce the clutter.

Example 25. Encryption with key `k`, omitted nonce `n`, associated data `ad`, and plaintext `plaintext` is performed as

```
M = ENC(k, plaintext)
macM = HASH({k, ad, plaintext})
```

This produces the ciphertext `M` and authentication data `macM`. The corresponding decryption is performed as

```
plaintext = DEC(k, M)
EQUALS(macM, HASH({k, ad, plaintext}))?
```

3.3.2 Protocol specification

The complete protocol specification is too long to be explicitly stated here. We provide it in the files associated with this thesis. The protocol specification contains repetitions of very similar parts. The only thing that changes are the keypairs used in these parts. We demonstrate how each part can be specified using an example.

Every Noise protocol maintains 3 values besides the DH keypairs. The key `k` is used as the actual key for encryption. The chaining key `ck` is used for deriving the values of `k`. The chaining key is initialized using constant values that describe the specification of the protocol. We initialize it just as a constant `'ck0'` because it does not affect security once the parties agreed on the protocol that is used. The hash `h` is used as associated data for encryption. It comprises all the messages that were sent during the protocol and the initial assumptions. A protocol is specified as a handshake pattern. We describe how to interpret the handshake patterns using the pattern that we are analyzing, the IK pattern.

```
<- s
...
-> e, es, s, ss
<- e, ee, se
```

Each arrow below the dots specifies a single message. The arrow above the dots specifies initial assumptions – the distribution of a public key. Alice is the party on the left side of the arrows and Bob is the party on the right. For example, `<-` specifies a message from Bob to Alice.

Single-character tokens represent actual transmissions of public keys. They are `e` for ephemeral key and `s` for static. If a key was already established (if a double-character token precedes them) then they are sent encrypted. Otherwise, they are sent in plaintext. The transmitted message is hashed together with `h` to produce a new value for `h`.

Example 26. The following code represents the processing the first token `e`.

```
Alice [
  generates ESA
  EPA = 'g'^ESA
  h1 = HASH({h0, EPA})
]
Alice -> Bob: EPA
Bob [
  h1 = HASH({h0, EPA})
]
```

The first token `s` comes after a double-character token, therefore it is encrypted. It is a static key, so it is known by Alice at the start of the protocol. The following code represents how it is processed.

```
Alice [
  knows private SSA
  SPA = 'g'^SSA
  Mspa = ENC(k1, SPA)
  macMspa = HASH({k1, h1, SPA})
  h2 = HASH({h1, Mspa})
]
Alice -> Bob: {Mspa, macMspa}
Bob [
  SPA = DEC(k1, SPA)
  EQUALS(macMspa, HASH({k1, h1, SPA}))?
  h2 = HASH({h1, Mspa})
]
```

Double-character tokens represent a DH calculation (without any transmission). The first character specifies the keypair from Alice and the second specifies the keypair

from Bob (ephemeral or static). The results of this calculation are used to derive new values for the key and the chaining key. These new values are derived using a sequence of hashes. We provide an example instead of a verbal description of the process.

Example 27. The following code represents the processing of the `es` token by both parties. Both parties derive the new key and chaining key the same way. The DH calculation has to be performed using the secret key available to each party.

```
Alice [
  DHes = SPB^ESA
  ck1 = HASH({HASH({ck0, DHes}), '1'})
  k1 = HASH({HASH({ck0, DHes}), ck1, '2'})
]
Bob [
  DHes = EPA^SSB
  ck1 = HASH({HASH({ck0, DHes}), '1'})
  k1 = HASH({HASH({ck0, DHes}), ck1, '2'})
]
```

Every handshake message (every line after the dots) is followed by an encrypted payload. The payloads are encrypted the same way as public keys. The messages are also hashed to produce a new value for the associated data.

Example 28. The following code represents processing the first payload message by Alice.

```
Alice [
  generates payload1
  M1 = ENC(k2, payload1)
  macM1 = HASH({k2, h2}, payload1)
  h3 = HASH({h2, M1})
]
```

3.3.3 Security properties

We verify the security properties of messages sent by Alice that should hold according to [13]. We examine the message payload sent alongside the first handshake message and payloads sent after a successful handshake.

The first payload message is encrypted to a known recipient. This means that it should be confidential in our model. We specify the following query to verify this property.

```
confidentiality? Alice's payload1
```

Its forward secrecy is guaranteed only for the sender compromise. Our forward secrecy query allows the adversary to compromise the long-term values of both parties. This means that the forward following forward secrecy query should not hold.

```
forward-secrecy? Alice's payload1
```

The protocol provides authentication of the ownership of the static key transmitted in the first handshake message. It does not provide authentication of the actual principal Alice. The adversary may frame the messages for Bob using his own keypairs. For that reason, the following query should fail.

```
authentication? Alice -> Bob: M1
```

The messages after the handshake are guaranteed strong forward secrecy. We verify this property using the following query.

```
forward-secrecy? Alice's payload3
```

Alice knows that `SSB` belongs to Bob as part of the initial assumptions. Authentication of the ownership of this key is equivalent to the authentication of Bob himself. The property is also injective because the message is encrypted using a key derived from Alice's ephemeral keypair. This allows us to verify this property using the following query.

```
authentication? Bob -> Alice: M2
```

All of the queries produce the expected results. The analysis produces logging messages that our program cannot parse yet. The results are parsed and translated, but we do not include them because of their size and complexity.

3.4 Assisting the Tamarin-Prover loading procedure

This case study demonstrates the process of adding optional sorts to variables in order to reduce the loading times of Tamarin-Prover. Tamarin-Prover performs precomputations on protocols that improve efficiency of the analysis of security properties. This process normally takes several seconds or less. However, if the protocol uses exponentiation in certain ways, it may take excessive amounts of time. The problem seems to occur when a single term is constructed using multiple Diffie-Hellman operations. In many cases, the protocol can be altered to make these loading times manageable.

We have created a minimal example protocol that does not load within one hour on our testing setup. The original and altered version of the protocol can also be found in the files associated with this thesis. We have not found out how long exactly is the loading time of the original protocol. Our analysis of a Noise protocol from section 3.3 uses Diffie-Hellman operations to a much greater extent. This analysis would not be possible unless we can reduce the loading time of the minimal example to a negligible amount.

The protocol was derived from the Noise protocol and uses similar naming conventions as section 3.3. Most notably, the variable identifiers consist of three characters.

First for ephemeral or static, second for secret or public, and third for Alice or Bob. The protocol performs two Diffie-Hellman calculations and uses the result to exchange a single message. The protocol specification consists of three short rules, one for initialization and one for each party. The following code shows the entire protocol.

```
theory spthy
begin
builtins: symmetric-encryption, diffie-hellman

rule init: let
  SPA = 'g'^~SSA
  SPB = 'g'^~SSB
in [
  Fr(~SSA),
  Fr(~SSB)
  ]-->[
  !Alice_init($Alice, SPA, SPB, ~SSA),
  !Bob_init($Bob, SPA, SPB, ~SSB),
  Out(SPA),
  Out(SPB)
]
rule Alice_1_0: let
  EPA = 'g'^~ESA
  DHes = SPB^~ESA
  DHss = SPB^SSA
  M = senc(~payload, <DHes, DHss>)
in [
  !Alice_init(Alice, SPA, SPB, SSA),
  Fr(~payload),
  Fr(~ESA)
]-->[
  Out(EPA),
  Out(M)
]
rule Bob_2_1: let
  DHes = EPA^SSB
  DHss = SPA^SSB
in [
  !Bob_init(Bob, SPA, SPB, SSB),
  In(EPA),
  In(senc(payload, <DHes, DHss>))
]-->[]
end
```

We can improve the loading time of this protocol by carefully adding sort prefixes to variables. We can afford to add these prefixes only if it does not change the semantics

of the protocol.

The initial rule adds variable `SSA` to Alice's initial state as fresh (prefixed with `~`). Afterwards, Alice's rule retrieves it unmarked. We know that this variable represents a key that was generated by Alice. This means that Alice is certain that it cannot be a public value. We can safely mark it with the `~` prefix in Alice's rule as well.

This is more complicated for the variable `payload`. Alice generates it in her rule. Even if she had generated it in the initialization rule, it could be marked as fresh in her rule. Bob learns this variable (a variable derived from it) in a message. Bob cannot verify that `payload` is not public or constructed from other terms. As such, it must remain without a prefix in his rule.

In total, we can mark `payload` in Alice's rule, `SSA`, and `SSB` with the prefix `~`. The identity variables `Alice` and `Bob` can be prefixed with `$`. The protocol loads within several seconds after adding these changes.

All of the changes are applied to variables that also appear in a state fact in the premises of the corresponding rule. These state facts are only created by a single rule. Within this rule, the sorts of variables are the same as the sorts we just added. This means that the addition of sort prefixes, as we describe it, does not change the semantics of the protocol at all. This inference of sorts can also be performed directly on the specifications in the Tamarin-Prover input language.

Conclusion

This thesis introduces a simplified language for specification of cryptographic protocols and a tool that verifies their security properties using Tamarin-Prover. We argue that our input language is more intuitive and easy to use compared to the input language of Tamarin-Prover. Protocol specifications are closer related to the real-world implementation. We use descriptive English statements instead of abbreviations and special symbols. The language does not require the user to manage the knowledge of principals. It features a clever manipulation with messages thanks to a special syntax for tuple deconstruction and scopes for variables. This approach also has its limitations. Our language is much less expressive. It only allows specifications of protocols that consist of supported elements.

Our input language and compiler prevent a wide range of common user errors. Our notion of expected equality prevents specification of a large class of protocols that are not meaningful in real-world applications. The expected equality of terms cannot be verified on protocol specifications in the input language of Tamarin-Prover. Additionally, our language uses predefined cryptographic functions and security properties because user-defined components of protocols are often the source of errors.

We introduce a procedure that adds optional sort prefixes to variables. This procedure reduces the loading times of protocols in Tamarin-Prover. Using this procedure lets us analyze some protocols for which the analysis would otherwise be effectively nonterminating. This improvement could be performed even on protocol specifications in the Tamarin-Prover input language, but Tamarin-Prover itself does not implement it. Our original intention was to create source lemmas, a feature of Tamarin-Prover that guides the analysis to completion. We did not follow up on this idea due to time constraints, but it may be an interesting point of further research.

We attempted to translate the text output of Tamarin-Prover to produce results in a clear and comprehensible form. This required extending the output of Tamarin-Prover by minor changes to its source code. We recognize that this effort did not yield satisfactory results. The output of our implementation is incomplete and ambiguous in cases where the text output of Tamarin-Prover is still insufficient. Furthermore, understanding the output requires a great deal of effort end experience. We believe that translating the interactive interface of Tamarin-Prover would have been more

successful, but also very work-intensive.

We demonstrate the correctness of our implementation by analyzing several protocols. The main purpose of our tool is to analyze old protocols with textbook attack examples. In cases of old protocols, we show examples where our tool is capable of discovering the known attacks that exploit their weaknesses. The files associated with this thesis also contain protocols for which the analysis does not terminate. Nontermination in some cases has to be accepted because the analysis of the properties of cryptographic protocols is an undecidable problem. We also demonstrate that our tool is capable of analyzing complex, modern protocols by analyzing a Noise protocol. This shows that despite its limitations, our language is sufficiently expressive to specify and analyze a reasonable class of protocols.

Bibliography

[1] Ross Anderson. The Classification of Hash Functions, 1993. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.1096.

[2] Bruno Blanchet. Security Protocol Verification: Symbolic and Computational Models. In *Principles of Security and Trust*, pages 3–29. Springer Berlin Heidelberg, 2012.

[3] Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif. In *Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2014.

[4] Michael Burrows, Martin Abadi, and Roger Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 1990.

[5] John Clark and Jeremy Jacob. A Survey of Authentication Protocol Literature: Version 1.0, 1997.

[6] Nachum Dershowitz and Jean-Pierre Jouannaud. *Rewrite Systems*, page 243–320. MIT Press, Cambridge, MA, USA, 1991.

[7] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[8] Michel Keller and David Basin. Converting Alice&Bob Protocol Specifications to Tamarin. Bachelor's thesis, ETH Zurich, 2014. https://infsec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/thesis_keller_alicebob.pdf.

[9] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic Protocol Analysis for the Real World. Cryptology ePrint Archive, Report 2019/971, 2019. https://eprint.iacr.org/2019/971.

[10] Gavin Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer Berlin Heidelberg, 1996.

[11] Simon Meier. *Advancing automated security protocol verification.* PhD thesis, ETH Zurich, 2013.
https://www.research-collection.ethz.ch/handle/20.500.11850/66840.

[12] Dave Otway and Owen Rees. Efficient and Timely Mutual Authentication. *ACM SIGOPS Operating Systems Review*, 1987.

[13] Trevor Perrin. The Noise Protocol Framework, 2018.
https://noiseprotocol.org/noise.html.

[14] Benedikt Schmidt. *Formal analysis of key exchange protocols and physical protocols.* PhD thesis, ETH Zurich, 2012.
https://www.research-collection.ethz.ch/handle/20.500.11850/72713.

[15] The Tamarin Team. *Tamarin-Prover Manual*, 2021.
https://tamarin-prover.github.io/manual/index.html.