Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis

Formalising Zero-Knowledge Proofs in the Symbolic Model

Sabina Fischlin

8 September 2021

Supervisors: Dr. Ralf Sasse, Dr. Dennis Jackson
Professor: Prof. Dr. David Basin

Department of Computer Science, ETH Zürich

Abstract

With most of our everyday life taking place online – from private conversations in messaging applications to financial transactions even being published on a public ledger – security properties such as anonymity and privacy are in ever higher demand. These properties typically require cryptographic schemes beyond encryption and traditional signatures. Zero-knowledge proofs often provide an elegant and powerful way in which anonymity and privacy can be achieved. Because of this, zero-knowledge proofs are now ever more frequently employed in security protocols.

As with all security protocols, one wants a way to prove that the desired properties are indeed fulfilled, since there are often unexpected attacks. In the past decades there have been a lot of advances in the automated analysis of security protocols where their properties are proven in the symbolic model. Tools have been developed, such as the Tamarin prover and ProVerif, which support an automated analysis. While initially these proofs have focused mainly on protocols using hashing, encryption and signing, recent work has introduced more complex cryptographic primitives to the symbolic model, such as zero-knowledge proofs. However, zero-knowledge proof models are still rather rare and differ vastly, often unexplicably so, and – to our knowledge – no zero-knowledge proofs have yet been modeled in Tamarin.

We have conducted an in-depth analysis of three existing zero-knowledge proof models and introduce a model for the Tamarin prover. We have applied this model to various simple examples and a real-world protocol: the Direct Anonymous Attestation (DAA) protocol, for which we were successfully able to prove several security properties.

In addition to our initial model, we present two possible, in some cases more beneficial, alternatives. We conduct a clear analysis in which we compare not only our own model and its alternatives, but also the already existing models. We thus provide a solid basis and pave the way for the analysis of further zero-knowledge proof protocols in Tamarin.

Acknowledgments

I first and foremost would like to thank my supervisors, Ralf Sasse and Dennis Jackson, for their time, invaluable insights and especially their flexibility. As my thesis was disrupted by a rather special life event – the birth of my son – I am very grateful for the support that I received not only from Ralf and Dennis, but also Denise Spicher and the team of the Registrar's Office. My thanks go also to David Basin for the opportunity to write this thesis in his group.

Contents

1 Introduction

As we conduct more and more of our everyday, private life online, anonymity and privacy are becoming ever more important security properties. This has become prominent for example in blockchain protocols, where the entire ledger containing all (private) transactions is publicly accessible. It is therefore an understandable desire that anonymity should at least be guaranteed in face of such public information. Another example is in messaging applications, where we likely want our conversations to remain private and anonymous, even to the server which authenticates the communicating parties.

However, both anonymity and privacy are not the easiest security properties to achieve and they typically require cryptographic schemes beyond traditional ones like encryption and signatures. Instead, these kinds of properties can often be achieved through zero-knowledge proofs. For example, a protocol was presented in [1] which enables anonymous authentication in the Signal application through the use of zero-knowledge proofs amongst other cryptographic schemes.

Zero-knowledge proofs were first introduced in 1985 by Goldwasser et al. in [2]. At a very high level, they provide a mechanism to prove the validity of a public statement without revealing any additional information except that the statement is true. In particular, this includes the actual proof *why* it is true in the first place. This gives zero-knowledge proofs a rather counter-intuitive nature but also makes them a very powerful tool: if the proof itself contains sensitive data, like an identifier or a pseudonym, it can be successfully kept secret. Because of this, such difficult properties like anonymity can even be achieved.

We will later provide several simple examples to illustrate this counter-intuitive property of zero-knowledge proofs before moving on to more complex ones, such as our case study.

Since zero-knowledge proofs provide such an elegant but powerful way to achieve these difficult properties, they are being more frequently employed in real-world protocols. But as we know, there are often unexpected attacks on protocols which break these desired security properties (see for example [3]). This means that we need a way to verify whether or not the protocol fulfills these properties.

Much reasearch has already been done in the field of protocol analysis in general, typically focusing on protocols using encryption and signing. Since manual verification has long been known to be error-prone – the distributed nature of protocols and the strong capabilities of the adversary, mean that vulnerabilities are often missed – a lot of work has been put towards automating the verification process. In order to do this, protocols are typically formalised in the symbolic model on which properties can be mathematically reasoned about. Symbolic models provide a layer of abstraction from the real world as they make strong assumptions about the behaviour of cryptographic primitives[1], in particular that they are perfectly secure in a cryptographic sense. This then allows us to automate this verification process. Various tools exist which facilitate the automated analysis of security protocols, such as Tamarin [4] and ProVerif [5].

As briefly mentioned above, initial analyses have mainly focused on protocols using encryption and signing primitives. However, with the growing use of zero-knowledge proofs in real-world protocols, it becomes important that we can also conduct an automated analysis on these kinds of protocols. Zero-knowledge proofs have first been introduced to the symbolic model in [6]. But, zero-knowledge proof models are still quite rare and also seem to differ vastly in their design and therefore also their limitations. This is in stark contrast to other cryptographic primitives, e.g. encryption, for which the models hardly ever differ.

A natural question is therefore where these differences come from and what the challenges are. The main crux in the modeling of zero-knowledge proofs seems to be with regards to what actually constitutes a valid statement. Somewhat unsurprisingly, this is specific to each individual proof. A further question is therefore whether or not it is possible to provide a generic solution to this problem.

Existing models have been implemented in ProVerif and have used the CoSP framework with a strong focus on providing computational soundness of the model. As far as we know, there exists no model so far for the Tamarin prover. With Tamarin being a state-of-the-art analysis tool, a model which perhaps could even exploit the particular functionalities which Tamarin provides would be useful.

We therefore conduct an in-depth analysis of three such models found in [6], [7] and [8], before we present our own zero-knowledge proof model for Tamarin.

Outline and contribution

We first provide a thorough basis of the underlying theory in section 2. We will in particular discuss symbolic models, term rewriting – which enables a formalisation of security protocols – and zero-knowledge proofs. We will also provide an overview of related work.

As we mentioned, some work on the symbolic modeling of zero-knowledge proof has already been done. In section 3, we will give an in-depth analysis on three existing models introduced in [6], [7] and [8].

We then apply our insights gained from this analysis to the design of our own model of zero-knowledge proofs in the Tamarin prover, which we will present in section 4. In section 5 we will apply this model to our case study, the Direct Anonymous Attestation protocol.

[1]e.g. encryption, hashing and signature schemes

As we have come across some limitations of our initial model, we will then introduce two possible modifications of this model in sections 6.1 and 6.2 and discuss potential advantages and disadvantages of each of these modifications in section 6.3.

We then conclude by comparing our results to previously discussed existing models in section 7 and give an overview of open questions and future work in 8.

Notation

We generally denote the tuple $(x_1, ..., x_n)$ as $\underline{x}$. Mathematical functions will be written in typical math notation, i.e. $func(t_1, ..., t_n)$. When we denote function symbols which will be used inside a protocol model, we will instead write this as $\mathsf{func}(t_1, ..., t_n)$. Then as soon as we refer to the actual implementation in the code, we will use `func(t1,...,tn)` and the correct syntax in order to highlight that distinction.

Once we have defined the parameters of a function, we will often abbreviate the full definition of the function, i.e. $func(t_1, ..., t_n)$ can be abbreviated to $func(.)$, and analogously $\mathsf{func}(t_1, ..., t_n)$ can be written as $\mathsf{func}(.)$.

An exception to these conventions occurs in section 3.1, where we present an overview of already existing models from other papers and follow the authors' notation as closely as possible. In this case, we will give a detailed explanation of the notational differences.

2 Background

In this section we introduce various concepts and definitions regarding symbolic models, zero-knowledge proofs and review the previous work which has been done on this topic.

2.1 Symbolic models

The definitions in this section are taken from [9], [10], [11], [12], [13], [14] and [15], unless stated otherwise.

2.1.1 Cryptographic primitives, symbolic models and computational soundness

Security protocols typically use cryptographic primitives – i.e. low-level cryptographic algorithms such as secure hashing or encryption functions – in order to guarantee certain security properties, even in the presence of an attacker. While cryptographers prove the security of the isolated primitives, using them inside a protocol oftentimes leads to attacks that are unexpected given the security properties of the primitives themselves.

In [3], the authors give an example of a security protocol in which a message is asymmetrically encrypted and exchanged between two parties in such a way that the message remains secret. However, by simply encrypting the message twice instead of once, the secrecy property is, rather surprisingly, broken.

It is therefore crucial to not only analyse the security properties of the cyrptographic primitives, but also the protocols themselves. Any formal analysis of a security protocol needs to specify the claimed properties and the assumptions under which these claims hold. A commonly used set of properties and assumptions is referred to as a model. Two common such models are the *computational* and the *symbolic* model.

Using the **computational model** we can prove security properties manually or using semi-automated tools. In this model, the messages exchanged are represented by bitstrings and the cryptographic primitives by functions from bitstrings to bitstrings. The adversary is then considered to be any probabilistic Turing machine.

While we can make strong statements in the computational model since we only make few assumptions, such proofs requiring vast extent of human-involvement are error-prone and it can be hard to precisely define the security properties one wants to prove. We therefore instead may want to perform these proofs in the symbolic model.

The **symbolic model** for protocol analysis was first introduced in [3] and is therefore sometimes referred to as the *Dolev-Yao model*, named after its authors. In this model we do not consider computational attacks and instead use a logical abstraction of cryptographic primitives and protocol interactions, i.e. we use a symbolic representation. In particular, we represent cryptographic primitives as function symbols and the messages sent in the protocol as terms composed of these symbols.

This of course means that we have to make somewhat stronger assumptions than in the computational model. We in particular make the assumption that we have *perfect cryptography*, i.e. it is impossible for the adversary or any other protocol participant to undermine the security definitions of the cryptographic operations. This is in fact what enables us to reason about protocols symbolically.

We also assume a so-called *Dolev-Yao adversary* which was first introduced, as its name suggests, by Dolev and Yao in [3]. This adversary is in complete control of the network, i.e. he can eavesdrop on messages as well as intercept, modify and fake messages. We can understand this as every message being sent to the adversary and he can then decide whether to withhold it, send it out as is or construct a different message for which he may use anything previously sent on the network. He is also a legitimate user of the system and can therefore communicate with any other user. In the context of protocols, we will refer to any user as *agent*.

We should also be aware that with all information represented as terms, an agent can only know a term in its entirety, i.e. it is not possible to know parts of a term or extract even a single bit. Additionally, we also require an unbounded number of sessions, which means that the protocol may be executed an arbitrary number of times and between an arbitrary set of agents.

While the requirement of stronger assumptions can be seen as a downside to the symbolic model, it also comes with significant advantages. Most importantly, the symbolic analysis enables us to perform proofs automatically instead of by hand. This drastically reduces the risk of human-error.

It is natural to ask whether a proof made in the symbolic model also holds in the computational model. This is referred to as *computational soundness*. It is not always clear if a particular symbolic model is computationally sound and proving this property might even be quite cumbersome. As we will see in section 2.3, in some cases ensuring computational soundness may even limit the scope of the model.

However, it is also not always crucial to know whether or not this property holds: in the computational model we capture all possible cryptographic attacks, but they might not be of the biggest interest for the construction of the protocol. For these reasons it should be considered whether it is beneficial to forgo the certainty of computational soundness for speed and ease of use. We do want to mention at this point that for precisely these reasons we do not consider computational soundness in the scope of this thesis. Naturally, this means however that there could be some limits with regards to the attacks our model can find and, more importantly, the strength of the proofs we provide.

As we have mentioned, analysing protocols in the symbolic model means that we can defer verification of security properties to a tool. One such tool is the Tamarin prover, which we will look at more closely in this chapter.

2.1.2 Term rewriting and mathematical background

As we have discussed in 2.1.1, in the symbolic model, cryptographic primitives, messages and indeed any information are represented as *terms*. We will now give a brief introduction to the mathematical background of terms, term algebras, equational theories and term rewriting which are used to represent security protocols, model the adversary's capabilities and can ultimately achieve automated protocol verification.

We want to stress that this is a rather large topic and that we only give a brief summary. For an in-depth treatment on term rewriting, we refer the reader to [9].

Terms are nested constructs built from function symbols, variables and other terms. As an example, let us say f is an n-ary function and $x_1, ..., x_n$ are variables, then $f(x_1, ..., x_n)$ as well as $x_1, ..., x_n$ are terms. We can then also specify which function symbols are applicable to a certain context, which in our case is a specific security protocol. This is done by defining a *signature*.

Definition 2.1 (Signature)**.** A **signature** Σ is a set of function symbols, where each $f \in \Sigma$ is associated with an integer $n \geq 0$, the **arity** of f. The set of all n-ary elements of Σ is also denoted by $\Sigma^{(n)}$. We call function symbols of arity 0, i.e. elements of $\Sigma^{(0)}$, **constants**.

For the following definitions we assume the two disjoint, countably infinite sets of **variables** $\mathcal{V}$ and **names** $\mathcal{N}$. Names are sometimes also referred to as constants, however as we mostly want to keep $\mathcal{N}$ distinct from Σ, these are not to be confused with the 0-ary terms from the previous definition.

Definition 2.2 (Term algebra)**.** Let Σ be a signature and $\mathcal{X} \subseteq \mathcal{V} \cup \mathcal{N}$ a set of variables and constants, where $\mathcal{X}$ and Σ are disjoint. We call the set $\mathcal{T}(\Sigma, \mathcal{X})$ a **term algebra** which is inductively defined as:

1. $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$
2. $\forall n \geq 0,\ f \in \Sigma^{(n)}$. If $t_1, ..., t_n \in \mathcal{T}(\Sigma, \mathcal{X})$, then $f(t_1, ..., t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$

We have already briefly introduced the informal notion of terms, but we can now define them formally.

Definition 2.3 (Term and ground term)**.** An element of $\mathcal{T}(\Sigma, \mathcal{X})$ is called a **term**. We can also define the set of **ground** terms as $\mathcal{T}(\Sigma, \mathcal{N})$, i.e. all terms which have been constructed without variables.

Definition 2.4 (Position and subterm)**.** A **position** p in a term t is a finite sequence of positive integers. The empty sequence is denoted by []. The **subterm** $t|_p$ of t at position p is defined as:

1. for any term t: $t|_{[]} = t$

2. for any $t = f(t_1, ..., t_n)$ with $f \in \mathcal{T}(\Sigma, \mathcal{X})$ and $p = [i] \cdot p'$ where i is a positive integer with $1 \leq i \leq n$ and p' a sequence: $t|_p = t_i|_p$

If none of these two cases apply, $t|_p$ does not exist.

A position therefore enables us to refer directly to a subterm inside a term. For example if $t = dec(enc(m, k), k)$, then $t|_{[1,2]} = k$.

We additionally want to introduce the following notation: we write $t[u]_p$ to denote the term t in which the subterm at position p, i.e. $t|_p$, has been replaced by the term u.

Definition 2.5 (Substitution)**.** A **substitution** is a function $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{X})$, such that $\sigma(x) \neq x$ for a finite number of variables $x \in \mathcal{X}$. We write a substitution of the variable x with the term t as $\{x \mapsto t\}$. Any substitution σ can be extended to a mapping $\hat{\sigma} : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{T}(\Sigma, \mathcal{X})$ where:

1. for variables $x \in \mathcal{V}$: $\hat{\sigma}(x) := \sigma(x)$
2. for non-variable terms $s = f(s_1, ..., s_n)$: $\hat{\sigma}(s) := f(\hat{\sigma}(s_1), ..., \hat{\sigma}(s_n))$

Using these definitions we can now move towards the definitions of equational theories and with that term rewriting. This is of particular interest when it comes to the formalisation of protocols since what we want to represent is what knowledge the adversary can gain from the messages which are sent over the network (or from compromising an agent). As we will see, this capability can be represented through equational theories and term rewriting. In other words, by stating which terms are equal to each other, or rather which can be reduced to another, we can represent the knowledge gain of the adversary.

Definition 2.6 (Equational theory)**.** An **equation** over a signature Σ is denoted by $s \approx t$ where $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$. We call a set of such equations E over Σ the **equational theory** (Σ, E). The set E induces a relation $\approx_E$ which is defined as the smallest congruence on Σ and contains all instances of equations of E. This gives rise to the **equivalence class** $[t]_E$ of a term t modulo E.

In order to determine which terms are equivalent under an equational theory, we use term rewriting. Informally, term rewriting enables us to reduce a term to another one according to the equations in the theory. If we can then reduce two terms to the same one, we consider them to be equal. We will now define this formally.

Definition 2.7 (Term rewriting)**.** A **rewrite rule** over a signature Σ is denoted by $l \to r$ where $l, r \in \mathcal{T}(\Sigma, \mathcal{V})$. A set of rewrite rules is referred to as a **rewrite system** R. We can define the corresponding **rewrite relation** $\to_R$, such that $s \to_R t$ if there is a position p in s, a rewrite rule $l \to r \in R$ and a substitution σ with $s|_p = \sigma(l)$ and $s[\sigma(r)]_p = t$.

Intuitively, we can understand the rewriting relation to denote that if a part of s "matches" the term l with σ then we substitute that part with a matching instantiation of r with σ.

Definition 2.8 (Termination, confluence and convergence)**.** A rewrite system R is called:

- **terminating** if there is no infinite sequence of terms $(t_1, t_2, ...)$ with $t_i \to_R t_{i+1}$
- **confluent** if $\forall t, s_1, s_2 \in \mathcal{T}(\Sigma, \mathcal{X})$ with $t \to_R^* s_1$ and $t \to_R^* s_2$, there exists $t' \in \mathcal{T}(\Sigma, \mathcal{X})$, such that $s_1 \to_R^* t'$ and $s_2 \to_R^* t'$, where $\to_R^*$ denotes the transitive closure of $\to_R$.
- **convergent** if it is both terminating and confluent.

If the rewriting system is convergent, then for each term there is a unique term on which no further rewrite rule can be applied. This term is defined as follows:

Definition 2.9 (Unique normal form)**.** Given a convergent rewriting system, then for all terms $t \in \mathcal{T}(\Sigma, \mathcal{X})$ there exists a unique term t', such that $t \to_R^* t'$ and no term t'' exists with $t' \to_R^* t''$.

Given an equational theory E, we can easily convert it into a set of rewriting rules R_E by assigning a direction to each equation. If the resulting system is convergent, we can effectively determine the equality of two terms, namely by comparing their unique normal forms in R_E.

We can now give a simple example of what these definitions mean with regards to a protocol specification. Let us consider the protocol where Alice sends Bob a nonce n encrypted with Bob's public key. Bob then replies with the hash of n. We can define a suitable signature as follows:

$$\Sigma := \{\mathsf{h}, \mathsf{pk}, \mathsf{sk}, \mathsf{enc}, \mathsf{dec}\}$$

The function symbols h, pk and sk have arity 1, enc and dec arity 2. The message Alice sends Bob would then be $\mathsf{enc}(n, \mathsf{pk}(B))$, whereas Bob replies with $\mathsf{h}(n)$. Our equational theory only contains one equation, namely:

$$\mathsf{dec}(\mathsf{enc}(m, \mathsf{pk}(X)), \mathsf{sk}(X)) = m$$

Intuitively, we can understand the equational theory to represent what the adversary – and any other protocol agent – can deduce from a message. Since we assume perfect cryptography, we do not specify any other equation, i.e. given the hash of a term, we cannot retrieve any information regarding the term and similarly for the encrypted nonce.

2.1.3 The Tamarin prover

The Tamarin prover is a tool to automatically analyse protocols in the symbolic model and was first introduced by Schmidt et al. in [11]. At a high-level, cryptographic primitives and their properties are modeled through a term algebra with an equational theory. The execution of the protocol, as well as the adversary's capabilities, are modeled as a *labeled multiset rewriting system*, which we will define further below. Messages sent on the adversary-controlled network (i.e. all messages are sent through the adversary) are modeled as terms, built from functions which must satisfy the underlying equational theory. The security properties are then specified in first-order logic as so-called *lemmas* and are defined over a protocol *trace*, which we will again define further below. Tamarin can then check the validity of these lemmas.

We now define some of the most crucial parts of the Tamarin prover, which are needed to understand the models we present in sections 4, 5 and 6. For details on the underlying theory, we refer the reader to [11] and [4], for details on the usage of Tamarin to the official manual [15].

Equational theories in Tamarin

Tamarin already comes with some built-in equational theories for – amongst others – asymmetric and symmetric encryption, signing, hashing and Diffie-Hellman exponentiation. In order to employ a built-in equational theory, the user needs to specify this after the keyword `builtins`. A full list and specification of the existing built-ins can be found in [15].

The user can also define their own equational theory. Function symbols are declared after the keyword `functions`. The syntax to declare a function symbol `f` is `f/n`, whereas n denotes the arity of `f`. Accompanying equations are declared after the keyword `equations`. As an example we can consider the following equational theory declaration for symmetric encryption[2]:

```
functions: enc/2, dec/2
equations: dec(enc(m, k), k) = m
```

In this thesis we often refer to a user-defined equation as an *equational rule* or in case of multiple rules an *equational rule set*.

The complete equational theory is then the union of the employed built-in equational theories and the user-defined equational theory.

Tamarin assigns an ordered **sort** to the terms of its term algebra. The topmost sort is denoted as *msg*. It also recognises the two distinct subsorts *fresh* and *pub*, denoting *fresh* and *public names*. Fresh names are only known to one agent, namely the one "creating" it, whereas public names are known globally.

Labeled multiset rewriting

As mentioned above, the protocol and its environment are modeled using a labeled multiset rewriting rules. These rules operate on the system's state,

[2]As previously stated there exists a built-in equational theory for symmetric encryption, which means that the user would not have to specify this themselves. This therefore merely serves as an illustrative example.

which is represented by a multiset of facts.

Definition 2.10 (Multiset)**.** A **multiset** m over a set X is a function $m : X \to \mathbb{N}$, where $m(x)$ denotes the multiplicity of $x \in X$.

Definition 2.11 (Fact)**.** Given a signature Σ of function symbols and another signature Σ_{fact} of so-called **fact symbols** with $\Sigma \cap \Sigma_{\text{fact}} = \emptyset$, the symbol $F(t_1, ..., t_k) \in \Sigma_{\text{fact}}$ of arity k is called a **fact** if $t_1, ..., t_k \in \mathcal{T}(\Sigma, \mathcal{X})$.

Definition 2.12 (Labeled multiset rewriting rule)**.** A **labeled multiset rewriting rule** is a triple (l, a, r). l, a and r are multisets of facts where l is referred to as the premises (or **left-hand-side**), r as the conclusions (or **right-hand-side**) and a as the **actions** (or labels). A multiset rewriting rule can be written as $l \xrightarrow{a} r$.

In the Tamarin syntax, a rewriting rule is written as:

```
rule name:
  [l]--[a]->[r]
```

Intuitively, an **execution** of a protocol can then be considered as a sequence of applied multiset rewriting rules – also called *rule instances*. A **protocol trace** can be considered as a sequence of action facts which appear at the rule instances of the protocol execution. An execution of course always starts from an empty state and then transitions to another state through a rule instance. A rule can only be applied if an instantiation of its premises is a subset of the current state. If a rule is applied, naturally the state of the system changes, making way for further rules to be applied.

We distinguish between different types of rules: rules to generate fresh names, *message deduction rules* which model the adversary's knowledge deduction throughout the protocol execution and rules which specify the protocol and additional adversary capabilities. We will not provide further details on the first two types of rules and instead refer the reader to [11]. Relevant to the user is the last rule type, which we can understand to comprise the following kinds of rules:

- *protocol rules* which specify the protocol itself
- *infrastructure rules* which provide auxiliary functionalities to the protocol, e.g. a public key infrastructure
- *adversary rules* which model additional adversary capabilities which are specific to the protocol, such as compromising an agent and thus extracting long-term private keys or other secrets. We naturally want the adversary to be as strong as possible. It is therefore crucial to include all appropriate user-defined adversary rules.

Facts

We recall that a state is represented simply by a multiset of facts. However, we must distinguish between two different types of facts, **persistent** and **linear**. A linear fact can only be used once in a state transition and will therefore be consumed by the application of an appropriate rule. This is not the case for persistent facts, which can be used an arbitrary amount of times. In Tamarin, persistent facts are denoted by a `!` in front of their name.

A user can define an arbitrary number of protocol-specific fact symbols, however Tamarin recognises the following special fact symbols:

- $\mathsf{K}(m)$ is a persistent fact which denotes that m is known to the adversary
- $\mathsf{Out}(m)$ is a linear fact which denotes that the protocol has sent a message m which can be received by the adversary
- $\mathsf{In}(m)$ is a linear fact which denotes that the advesary has sent a message m which can be received by the protocol
- $\mathsf{Fr}(n)$ is another linear fact denoting that n was freshly generated.

Lemmas

As previously mentioned, we can formulate the security properties of the protocol we would like to prove in first-order logic as so-called **lemmas** over not only terms of the sort *msg*, but also **timepoints** of action facts. As we consider a protocol trace as a sequence of action facts, an action fact's timepoint is then its position inside the trace. We write an action fact A at timepoint t_1 as $\mathsf{A}(.)@t_1$.

Restrictions

Another interesting feature of Tamarin are so-called **restrictions**. These enable us to limit the state space of protocol traces. Just as the security properties, they are written in first-order logic and basically restrict the set of traces considered in the protocol analysis to the ones satisfying the restriction. We will make extensive use of restrictions in our own models presented in sections 4, 5 and in particular 6.1.

Beyond Tamarin

The Tamarin prover is of course not the only tool which exists for automated protocol verification. Another commonly used tool is ProVerif. We will not introduce ProVerif in detail and refer the reader instead to its user manual [5]. For the verification of our own models in sections 4, 5 and 6 we use the Tamarin prover.

2.2 Zero-knowledge proofs

As stated in section 1, zero-knowledge proofs were first introduced in 1985 by Goldwasser et al. in [2] and have since found wide applications such as in anonymous signature schemes and blockchain protocols. Intuitively, zero-knowledge proofs are used to prove the validity of a statement without revealing any additional information besides the fact that the statement is true – including the reason why the statement is true.

This seems almost like a contradictory statement, which contributes to the fact that there is a somewhat counter-intuitive nature to zero-knowledge proofs. But this zero-knowledge property also makes them incredibly powerful and useful in protocols where anonymity and/or privacy are crucial and where information is exchanged between mutually distrustful parties.

There exist various different definitions of zero-knowledge proofs, adapted to the context in which they are presented. The definitions given in this section summarise the definitions found in [16], unless stated otherwise.

A simple example

Before going any further we want to introduce a simple example of a zero-knowledge proof[3]:

Let us imagine a setting where a cipher-text was encrypted using Alice's public key. Alice now needs to prove to Bob that she knows the corresponding secret key, obviously without either revealing the secret key nor decrypting the message. Nor does she want to solve this issue using a certificate as she wants to keep her identity secret and remain anonymous. In other words without revealing any further information beyond the fact that the statement that she knows the corresponding secret key is true.

We can phrase this in a more formal way (which we will formalise further in the following paragraphs):

Alice wants to prove to Bob that she knows the secret key which can decipher $c = \mathsf{aenc}(m, \mathsf{pk}(sk))$, where c refers to the ciphertext, m to the plain message, $\mathsf{pk}(sk)$ to the public key corresponding to the secret key sk and $\mathsf{aenc}(.)$ to an asymmetric encryption function.

It is common practice, in symbolic models, to define the public key as $\mathsf{pk}(Alice)$, i.e. bound to a certain agent. We deliberately define here however the public key only with regards to its corresponding secret key, i.e. $\mathsf{pk}(sk)$. This allows us to reason later on independently of any agents, which is exactly

[3]We highlight all parts in this section which define our example inside a box

what we want in order to achieve anonymity.

This simple example will be used as a running example throughout the thesis, especially in the following paragraphs as we move further towards a formal definition of zero-knowledge proofs.

Intuitive definition

A zero-knowledge proof typically requires (at least) two parties: a ***prover*** P who wants to convince one (or multiple) ***verifier***(s) V of the validity of a certain ***statement*** x[4]. V must decide to either reject or accept the statement.

As mentioned previously, the statement itself is naturally public. There must however always be a secret part to the proof, otherwise one would not require a zero-knowledge proof in the first place. This secret part is commonly referred to as the ***witness*** w, as it can be understood as bearing witness to the validity of the statement without actually revealing itself.

In the simple example introduced above, Alice is obviously the prover P and Bob is the verifier V. Alice's secret key sk serves as the witness.

Witness relation

Whether the verifier V rejects or accepts the validity of the statement x is determined by the ***witness relation*** $\mathcal{R}$[5]:

$$\mathcal{R} := \{(x, w) \mid \text{Predicates over x and w}\}$$

where the predicates define what constitutes a valid statement. V then accepts the statement x as valid (ideally) iff $(x, w) \in \mathcal{R}$. The witness relation can be understood as characterising the behaviour of the chosen zero-knowledge proof system. It is worth noting that any w which satisfies $(x, w) \in \mathcal{R}$ is considered to be a proof that x is a valid statement.

We can therefore define the witness relation of our simple example as:

$$\mathcal{R} := \{(x, w) \mid \exists m.\ x = \mathsf{aenc}(m, \mathsf{pk}(w))\}$$

[4]We will expand on what we mean by "at least two parties" when we introduce the distinction between interactive and non-interactive zero-knowledge proofs.

[5]In [16] the witness relation is actually defined as $\mathcal{R}_L$, i.e. associated with a language L. Proving that the statement x is valid, i.e. a true statement, can also be expressed as proving that $x \in L$. A zero-knowledge proof system is therefore always defined for a specific language L. L itself is defined as $L = \{x \mid \exists w.\ (x, w) \in \mathcal{R}_L\}$, i.e. x is considered to be valid if there exists a w, s.t. $(x, w) \in \mathcal{R}_L$. This was left out in our definition for simplicity reasons, as it is mainly relevant when arguing about which statements can indeed be proven by a zero-knowledge proof. This is outside of the scope of this thesis.

Interactive and non-interactive zero-knowledge proofs

Before presenting a formal definition of zero-knowledge proofs we must note that there is a distinction between an interactive and a non-interactive setting of such proofs.

When zero-knowledge proofs were first introduced in [2], they were interactive protocols, i.e. the verifier and the prover would send a series of messages back and forth. Typically, the verifier would toss a coin and ask the prover to give a response according to its result. This would be repeated multiple times. In [17] a non-interactive model was first introduced. The authors found that the randomness achieved through the interactive coin toss could be replaced by a *common random string* upon which the prover constructs his proof, i.e. some pre-agreed upon randomness used both in the construction and in the verification of the proof. This is commonly also referred to as a uniformly distributed ***common reference string***[6], or *CRS* for short [16].

In an interactive zero-knowledge proof the prover naturally can only interact and provide his proof to a single verifier. In the non-interactive setting the interaction is reduced to a single message which can be received by any or even multiple verifiers [6]. This is often referred to as a unidirectional interaction, since there is a single message being sent with regards to the proof from the prover to the verifier. One big advantage is therefore that it can even be processed further by the verifier, e.g. encrypted, or stored for future reference. This makes non-interactive proofs very powerful and opens up the door to further applications.

In our example, Alice wants to prove her statement primarily to Bob. But then Charlie requires a proof as well. In the non-interactive setting she can simply send the same proof to Charlie.

In this thesis we only consider the non-interactive setting. All definitions given in this chapter refer to a non-interactive zero-knowledge proof system.

Formal definition

In the following definition we denote $V(.) = 1$ as the verifier V accepting the input statement as valid. We additionally define $poly(|x|)$ as polynomial in the length of x.

Definition 2.13 (Non-interactive zero-knowledge proof system)**.** A ***non-interactive zero-knowledge proof system*** consists of a pair of probabilistic algorithms (P, V), i.e. the prover and the verifier, where V has to run in polynomial-time and the following three conditions hold (intuitive definition of each condition is annotated in brackets):

[6]A common reference string can have arbitrary distribution, whereas a common random string is a uniformly distributed common reference string [18],[16].

- **Soundness** (the prover cannot trick the verifier into accepting a false statement, i.e. a false statement is rejected by the verifier):
  For every false statement x and every possible prover B,

  $$Pr[V(x, CRS, B(x, CRS)) = 1] \leq 2^{-poly(|x|)}$$

  where CRS is a random variable uniformly distributed in $\{0,1\}^{poly(|x|)}$.

- **Completeness** (a true statement will be accepted by the verifier):
  For every true statement x,

  $$Pr[V(x, CRS, P(x, CRS)) = 1] \geq 1 - 2^{-poly(|x|)}$$

  where CRS is a random variable uniformly distributed in $\{0,1\}^{poly(|x|)}$.

- **Zero-knowledge** (Whatever the verifier can compute efficiently after interacting with the prover on the input x, they could also have computed efficiently by themselves without having received the proof. In other words, the proof yields no knowledge except for the validity of the statement x.):
  There exists a probabilistic polynomial-time algorithm M, s.t. the sets $\{(x, CRS, P(x, CRS))\}$ and $\{M(x)\}$, where CRS is a random variable uniformly distributed in $\{0,1\}^{poly(|x|)}$ and x is a true statement, are computationally indistinguishable.

In the literature, the probabilities on the right-hand side of the inequalities of the soundness and completeness properties are actually defined as $\frac{1}{3}$ and $\frac{2}{3}$ respectively. However, these probabilities can be decreased or increased to the values stated in the above definition by repeating the proving process sufficiently many times [16], [19]. For simplicity reasons, we assume that P always repeats this process multiple times and therefore already give the final probabilities in the definition above.

We will now give some additional information regarding these three properties and their meaning.

The soundness property has to hold for all potential provers B, while completeness must only hold for the particular prover P of the system. The zero-knowledge property may also be regarded as a property of the prover, i.e. its robustness against attempts to gain knowledge from it through the proof.

While we do not define the notion of knowledge mathematically in this thesis, we want to point out that gaining knowledge is related to computational difficulty. Intuitively, it can be understood that a party gains knowledge from an interaction iff they receive the result of a computation which is infeasible to them. In other words, they could not have made the same computation themselves. This is exactly what is stated in the zero-knowledge property.

The witness w and existential vs. knowledge soundness

It is important to note that in the definition above (2.13) the witness w is not present. This is linked to what we previously mentioned with regards to the witness relation: any witness w' which satisfies $(x, w') \in R$ is a proof that x is a valid statement. The validity of x is therefore not bound to a single w.

However, it can be understood that both P and V may use other sources of input. In particular, P may obviously make use of w, but also V may make use of additional (auxiliary) information z, which is only known to V. It is crucial to ensure that the zero-knowledge property holds even if there exists a dependency of z and w on x. It should also be noted that since P and V are probabilistic algorithms, they may also use additional randomness (beyond the CRS), which is again only known to them.

On a related note, in some literature, e.g. [20], a distinction is made between ***existential*** and ***knowledge soundness***. We will not give a formal definition here and instead refer the reader to [20]. Intuitively however, existential soundness can be understood to refer to the existence of a w, such that $(x, w) \in \mathcal{R}$, whereas knowledge soundness refers to the prover "knowing" a specific witness w' with $(x, w') \in \mathcal{R}$. Naturally, knowledge soundness implies existential soundness, since the prover cannot "know" a witness without the witness existing in the first place.

The soundness property given in definition 2.13 refers to existential soundness. However, for many protocols which use zero-knowledge proofs the more interesting property is that of knowledge soundness. In this thesis we will therefore mainly focus on the knowledge soundness property. When we refer to soundness, this is to be understood as knowledge soundness. In the rare case we make an argument regarding existential soundness, this will be named as such explicitly.

Multiple statement or witness terms

In our running example we introduced in this section, the statement x and the witness w are singular terms. However, there are other zero-knowledge proofs where both x or w consist of multiple terms. We will indeed see many such examples in section 4.3 and our case study which we present in section 5.

We can therefore expand the definitions given above by considering both the statement and the witness as tuples $\underline{x}$ and $\underline{w}$. In particular, we can define the witness relation as:

$$\mathcal{R} := \{(\underline{x}, \underline{w}) \mid \text{Predicates over } \underline{x} \text{ and } \underline{w}\}$$

While we might often refer to the statement as x and the witness as w, this can always be understood to also apply in case x and w are tuples.

Common reference string revisited

As we have seen in the formal definition 2.13, both P and V take the same common reference string as an argument. When we defined the witness relation and with it what constitutes a valid statement, we have said that the verifier

V accepts the statement x as valid iff there exists a w, such that $(x, w) \in \mathcal{R}$. While this obviously also holds in the non-interactive setting, we should add that the verifier can only accept a statement if it operates on the same common reference string as was used to create the proof. This is not important for any mathematical definition of valid statements, but we do want to mention that it does play a role in the symbolic model in order to argue about certain attacks regarding the common reference string.

We generally assume the common reference string to have been honestly generated [18], but apart from this assumption there are not as many requirements on this parameter as one might think. In [19], the authors state that in particular the zero-knowledge property of the proof does not depend on the secrecy nor even the unpredictability of the common reference string.

As was explored in [18], the requirement on the common reference string to have been honestly generated is not always a plausible one in the real-world. And indeed, some attacks are possible if the common reference string is malicious. The three properties specified in definition 2.13 therefore only hold under the assumption that the common reference string has been honestly generated. For this purpose, the authors in [18] define stronger security properties, such as *subversion soundness* and *subversion zero-knowledge*, which basically state that soundness and zero-knowledge respectively still hold in case of a malicious common reference string. What they have found is that some of these properties cannot hold simultaneously. In particular, it is not possible that subversion soundness and zero-knowledge both hold at the same time.

Additional properties

While we have only presented a handful of properties of zero-knowledge proofs so far, it is important to note that there exist many additional properties, including various different notions of zero-knowledge, some stricter and some weaker. In practice, it might be very interesting to look at other definitions of zero-knowledge as not many proof systems can fulfill the stronger properties. As we will see in later sections, our proposed model can be extended or modified to include the verification of additional properties. However, for this thesis we do not consider any properties beyond the three main properties soundness, completeness and zero-knowledge as defined in this section.

For definitions of other properties we refer the reader to [16], [20], [6] and [8].

2.3 Related work

Previous work has been done by Backes et al. with regards to formalising zero-knowledge proofs in the symbolic model, namely [6], [7] and [8]. With a similar group of authors working on all three papers, they build heavily on each other: [6] and [7] have been written around the same time, with the latter focusing on the implementation of a zero-knowledge proof model and the former focusing more on the underlying theory. [8] then weakens some of the assumptions made in [6]. Two of the models have been implemented, namely [7] and [8], the former in ProVerif and the latter inside the CoSP framework[7].

What is interesting to note is that while the papers are strongly connected, all three of them – rather surprisingly – propose vastly different models. We will take a closer look at these models and analyse them in-depth in section 3.

Zero-knowledge proofs were first introduced to the symbolic model in [6]. The authors put a lot of emphasis on ensuring computational soundness of their model. This has required them to define a set of properties that a zero-knowledge proof needs to fulfill in order to ensure computational soundness.

These properties go beyond the standard security properties defined in 2.2, i.e. soundness, completeness and zero-knowledge. It is not clear, nor was it investigated further, how many zero-knowledge proofs in practice actually fulfill these properties. This could potentially mean that the requirement to achieve computational soundness may limit the scope of the zero-knowledge proofs which can be modeled.

The authors themselves mention in [8] that to their knowledge there is only one zero-knowledge proof which satisfies the *extraction zero-knowledge* property (which is the same as a *non-malleability* property). In the paper following [6], i.e. [8], they therefore focused on finding a weaker condition especially with regards to the *extraction zero-knowledge* property. While all other properties still need to hold for computational soundness, they managed to reduce the *extraction zero-knowledge* to a *honest simulation-sound extractability* property. It must be mentioned, that in [8] the authors state that generic constructions exist which can transform any zero-knowledge proof into one fulfilling their (now weakened) requirements for computational soundness. This is quite interesting, we however did not investigate this further, as again, this lies outside the scope of this thesis.

While the authors do define a clear abstract model in [6], they do not mention whether or not they have implemented this. There exists however, an implementation in ProVerif of a slightly similar model outlined in [7], which was published around the same time. In this last paper, the authors in particular used their model to successfully verify the Direct Anonymous Attestation (DAA) protocol.

An ECC[8]-based variant of the DAA protocol has previously been modeled in Tamarin in [22] and [23]. While this is not related to zero-knowledge proofs, it is relevant for our work as we have also chosen to model the (zero-knowledge proof variant) DAA protocol as a case study in section 5.

As we will also see in our own design of a zero-knowledge proof model (see section 4), each zero-knowledge proof requires its own equational theory. The

[7]CoSP is a general framework which can be used to prove computational soundness of symbolic models and was introduced in [21].

[8]elliptic-curve cryptography

authors of [7] have therefore chosen to develop a compiler which encodes a zero-knowledge proof description into ProVerif specifications. This compiler has been used by Wang et al. to analyse a TTP[9]-free blacklistable anonymous credential system [24].

Even though both [6] and [7] were published around the same time and by the same authors, it is not clear how the conditions laid out in the former to ensure computational soundness impacted the implementation of the latter. There was no mention of any additional properties which had to hold in [7].

We want to stress that, as mentioned in section 2.1, achieving or ensuring computational soundness is not the goal of this work. We therefore are not concerned about the properties that have been identified neither in [6] nor in [8].

For completeness reasons, we also want to the mention the paper written by Backes et al. on malleable zero-knowledge proofs in the symbolic model [25]. Malleable zero-knowledge proofs are proofs which allow for certain kinds of transformations. In particular, an agent may re-randomize zero-knowledge proofs in order to for example logically compose existing zero-knowledge proofs. The authors were able to again prove computational soundness for their model. With the *extraction zero-knowledge* property from [6] and the *honest simulation-sound extractability* property from [8] begin closely related to *non-malleability*, this paper is particularly interesting as it seems to further weaken the requirements for computational soundness initially laid out in [6]. We will however not look at this paper further as it would unnecessarily expand the scope of this thesis.

[9]trusted third party

3 In-depth analysis of existing models

As mentioned in section 2.3, there exist mainly three papers which introduce symbolic models for zero-knowledge proofs: [6], [7] and [8]. While all three of these models have been introduced by a similar group of authors, they are very different. In this chapter we will analyse these differences in-depth and present possible reasons for these different approaches. We additionally try to understand what the advantages and disadvantages might be of each approach.

In order to conduct a thorough analysis, we tried to obtain a copy of the source code for these models, including the compiler developed in [7]. Unfortunately, we were unable to locate any such copy for any of the models neither by following links to online resources referenced by the papers nor by direct searches online. We have therefore made several attempts to contact the authors, but regrettably we did not receive a response. This means, in particular, that we were not able to replicate any of the verification results from the papers. The analysis presented here is therefore naturally only based on the content of the papers and may unfortunately not yield a complete picture, especially since the papers do not give much information regarding the actual implementation and how they solved the more tricky parts of the model.

This section is organised as follows: we first present a brief overview of the three models and their characteristics. We then attempt to standardise the models, i.e. put them in a form in which we can compare the various function symbols and equations. In the cases where the adversary's additional[10] deduction rules are specified, we offer a list of these rules. We then conclude with an evaluation of the models.

3.1 Overview of the models

For the overview of the three models, we have chosen to keep the original notation used by the authors in the respective papers as closely as possible. This may therefore deviate from the general notation otherwise used in this thesis. Whenever necessary we will therefore explain the particular notation used in the respective paper and give additional definitions if required.

[10]additional with regards to the already defined term equations

Backes et al. 2008 computational soundness paper [6]

In this model a zero-knowledge proof is defined as a function[11] $\mathsf{ZK}_F^R(\underline{r};\underline{a};\underline{b})$, whereas the term $\underline{t}$ is considered to be the tuple $(t_1, ..., t_n)$. We will now look at this function and its parameters in detail.

Central to their model of a zero-knowledge proof is the parameter F. This is a Boolean formula which – even though this was not specifically mentioned by the authors – can be understood to encode the predicates of the witness relation of the proof. The formula F is constructed over terms of the form $\mathsf{ZKTerm} = \mathsf{ZKTerm}$, whereas ZKTerm is defined as follows:

$$\mathsf{ZKTerm} = \mathsf{ek}(\beta_i) \mid \alpha_i \mid \beta_i \mid \langle \mathsf{ZKTerm}, \mathsf{ZKTerm} \rangle \mid \{\mathsf{ZKTerm}\}^{\rho_i}_{\mathsf{ek}(\beta_j)}$$

Here, the term $\mathsf{ek}(A)$ denotes the public key of an agent A, $\langle ., . \rangle$ a pair and $\{t\}^R_{\mathsf{ek}(A)}$ the encryption of a term t with the public key of A using randomness R. The α_i, β_i and ρ_i are the variable names which may be used in a ZKTerm. We will look at the distinction between α_i, β_i and ρ_i further down.

This introduction of ZKTerm can be understood as defining the proof language and with it establishes the scope of all the zero-knowledge proofs that the authors consider in their model[12].

We do want to point out that with the formula F being defined only over terms of the form $\mathsf{ZKTerm} = \mathsf{ZKTerm}$ as stated above, the zero-knowledge proofs which can be verified by this model are very limited. The only cryptographic primitive which may appear in the witness relation is an asymmetric encryption. Not even the corresponding asymmetric decryption is included. Given the definitions of the zero-knowledge proofs used in our case study protocol, which we present in section 5, we would not be able to verify that protocol in this, rather theoretical, model.

It should also be noted that there are no term equations defined in this model. In fact, the authors explicitly state that "no equational theory is involved". Instead, the authors define a set of deduction rules for the adversary only. We will take a closer look at these deduction rules in section 3.3.

The formula F should additionally clearly distinguish between random, secret and publicly known variables. The authors denote the secret variables by

[11]In the paper, this is referred to as a constructor and not a function. Oftentimes in ProVerif terminology, a distinction is made between constructor and destructor function symbols. The latter are ones for which equational rules exist, i.e. ones that can be rewritten. We will not make this distinction in this thesis and instead refer to both constructors and destructors as functions (or function symbols).

[12]We suspect that it represents all the zero-knowledge proofs for which the authors have proven their statements.

α_i, the random by ρ_i[13] and the public ones by β_i. Each zero-knowledge proof then comes with the actual values $\underline{r}$, $\underline{a}$ and $\underline{b}$ which will be assigned to these variables, i.e. these refer to the parameters in the zero-knowledge proof function $\mathrm{ZK}_F^R(\underline{r}; \underline{a}; \underline{b})$. The values $\underline{r}$, $\underline{a}$ and $\underline{b}$ are a bit more expressive as ZKTerm and may incorporate for example agent names, nonces, decryption keys (however still without a corresponding decryption function) and so-called "garbage" messages, which represent ill-formed messages.

After the replacement of ρ_i, α_i and β_i with their respective values from $\underline{r}$, $\underline{a}$ and $\underline{b}$, the formula F then either evaluates to *true* or *false* depending on these values. In addition to the set of the adversary's deduction rules we briefly mentioned above, this is what enables the authors to forgo the use of an equational theory in this model: by a simple replacement of the variables with $\underline{r}$, $\underline{a}$ and $\underline{b}$, the terms $\mathsf{ZKTerm} = \mathsf{ZKTerm}$ will either have the form $t = t$ – in which case the entire term can be replaced by *true* – or $t = u$ with $t \neq u$ – in which case the term is replaced by *false*. This equality is evaluated in a **strictly syntactical sense**. We are therefore left with a simple Boolean formula over *true* and *false* values which are then evaluated logically.

We will briefly illustrate this through the simple example introduced in section 2.2. As we recall, in this example, Alice wants to prove to Bob that she knows the secret key to a ciphertext in which the message was encrypted with the corresponding public key. The formula F can therefore be defined as:

$$F_{\mathrm{ex}} := (\beta_1 = \{\alpha_1\}^{\rho_1}_{\mathsf{ek}(\beta_2)})$$

The accompanying zero-knowledge proof would then be $\mathsf{ZK}^R_{F_{\mathrm{ex}}}(r; m; c, \mathrm{Alice})$, where $c := \{m\}^r_{\mathsf{ek}(\mathrm{Alice})}$. Replacing these values we then get the term $\{m\}^r_{\mathsf{ek}(\mathrm{Alice})} = \{m\}^r_{\mathsf{ek}(\mathrm{Alice})}$, i.e. a term of the form $t = t$, which trivially evaluates to *true* as stated above. We will look at the particularities of this model for our example further down.

The authors stress that they have chosen this notation, i.e. the clear separation of the formula F from the actual assignments $\underline{r}$, $\underline{a}$ and $\underline{b}$, deliberately. They mention that this makes it possible for them to differentiate easily and clearly between the public ($\underline{b}$) and secret values ($\underline{a}$ and $\underline{r}$). This advantage is clearly seen in their deduction rules, where the adversary may extract the public values $\underline{b}$, but has no ability to extract $\underline{r}$ or $\underline{a}$. We will inspect this further in section 3.3.

The parameter R from $\mathsf{ZK}^R_F(.)$ is still missing from our explanation. This refers to the abstract randomness that the probabilistic prover uses in his proof. This should not be confused with either the ρ_i used inside the formula F, nor the r_i values with which they are substituted. The ρ_i are used by the terms inside F, while R is used to construct the proof itself.

In summary, the function $\mathsf{ZK}^R_F(.)$ can be understood as the encoding of the witness relation inside a Boolean formula accompanied by the actual values with

[13]Whereas the random variables ρ_i are also secret, i.e. can be understood as a special kind of secret variables.

which the corresponding variables inside the formula are to be replaced.

Before we conclude our overview however, we briefly want to revisit the model we gave above for our simple example. We recall the corresponding witness relation, which we defined in section 2.2:

$$\mathcal{R} := \{(x, w) \mid \exists m.\ x = \mathsf{aenc}(m, \mathsf{pk}(w))\}$$

If we compare the formula F_{ex} we defined above for our example to $\mathcal{R}$, we can see that the formula does not entirely reflect the statement we want to make. For one, the secret key was supposed to be the witness, whereas in the above model the message m strangely has become the witness. We are struggling to represent this correctly because without a decryption function symbol or a way to somehow include a decryption key in a ZKTerm we simply cannot model Alice's secret key as the witness. However, even if we could make the secret key the witness, the message m would still either have to be part of the witness or the statement of the zero-knowledge proof, since the definition of ZKTerm does not allow for any variable names which will not be replaced later on. Another glaring difference between F_{ex} and $\mathcal{R}$ is that we had to bind the encryption key to the agent Alice: since the encryption key is defined as $\mathsf{ek}(\beta_i)$ and β_i refers to a *public* value, we could not use β_i to represent the secret key like we originally defined in the witness relation and instead had to replace β_i with the agent's identity, i.e. Alice. Including the agent's identity is precisely what we did not want to do in order to be able to establish anonymity. This in fact means that it is not the statement we wanted to prove. Instead, the statement we are proving is that the ciphertext c is a valid encryption of some message m with randomness r using the public key of Alice. In other words, we were actually not able to encode our simple example in this model!

We also find it worth pointing out that there was no reference made to the common reference string, which is an integral part of the definition of a non-interactive zero-knowledge proof (see definition 2.13 for details).

Even though this is the one model which is purely theoretical as it was not implemented by its authors, we still would have liked some information on how an implementation of this model could be achieved. A translation of this model to an (existing) automated tool does not seem straight-forward, especially with regards to the encoding of the formula F and its evaluation to either *true* or *false* according to a syntactical comparison of terms.

Backes et al. 2008 paper on zero-knowledge proofs in the applied pi-calculus [7]

The model presented in [7] is very similar to the one in [6]. There are however some small differences. We first give a brief overview of the model presented here and then point out the key differences to the model from [6].

A zero-knowledge proof is defined as a function $\mathsf{ZK}_{i,j}(\tilde{M}, \tilde{N}, F)$, whereas $\tilde{M}$ is described as the sequence of terms $M_1 ... M_i$ and $\tilde{N}$ as $N_1 ... N_j$. $\tilde{M}$ is described as the private component and $\tilde{N}$ as the public component. Similarly to the model in [6], F denotes a formula over the terms α_k and β_l where $k \in [1, i]$ and $l \in [1, j]$. The i and j in $\mathsf{ZK}_{i,j}$ therefore refer both to the arity of the function and the number of variables present in F. In contrast to [6], in this model, the verification process is modeled by the following equational rule:

$$\begin{aligned} &\mathsf{Ver}_{i,j}(F, \mathsf{ZK}_{i,j}(\tilde{M}, \tilde{N}, F)) = \mathsf{true}, \text{ iff} \\ &\qquad 1)\ E_{\mathsf{ZK}} \vdash F\{\tilde{M}/\tilde{\alpha}\}\{\tilde{N}/\tilde{\beta}\} = \mathsf{true} \\ &\qquad 2)\ F \text{ is an } (i,j)\text{-formula} \end{aligned}$$

Without going into too much detail regarding the semantics presented in the paper, the first condition states that the formula F, when substituting each α_k with M_k and each β_l with N_l, must reduce to true. We want to point out that in contrast to the model in [6], this refers to an equality with regards to an equational theory and not the value *true* in a logical sense.

The term (i, j)-formula in the second condition refers to a formula which only contains other functions defined in the equational theory, constants and the variables α_k and β_l. We will not give a full list here of the other functions they do define (instead we refer the reader to Table 4 in [7]), however their equational theory contains logical operators, a pairing function, asymmetric and symmetric encryption, signing, blind signing[14], hashing as well as the Boolean constants true and false.

As in the model from [6], F can be understood to encode the witness relation of the particular zero-knowledge proof, with the other paramters in $\mathsf{ZK}_{i,j}(.)$ then determining the actual values with which the variables inside the formula are to be replaced. A simple example of such a zero-knowledge proof function is given in the paper:

$$\mathsf{ZK}(k; m, \mathsf{enc}_{\mathsf{sym}}(m, k); \beta_2 = \mathsf{enc}_{\mathsf{sym}}(\beta_1, \alpha_1))$$

This example models the zero-knowledge proof that enc_{sym} is an encryption of m with k. In fact, this zero-knowledge proof is almost identical to our simple example introduced in 2.2, the only difference being that here the statement is regarding a symmetric encryption, instead of an asymmetric one as in our example. As mentioned above, the defined equational theory supports asymmetric encryption and decryption. We can therefore define the following zero-knowledge proof for our example (for details regarding the function symbols, we refer the reader to [6] or section 3.2):

$$\mathsf{ZK}(\mathsf{sk}(A); m, \mathsf{enc}_{\mathsf{asym}}(m, \mathsf{pk}(A)); \beta_1 = \mathsf{dec}_{\mathsf{asym}}(\beta_2, \alpha_1))$$

[14] Blind signing is necessary for their case study of the Direct Anonymous Attestation (DAA) protocol

This means that we are indeed able to encode our simple example in this model. Just as in the model from [6], we did have to make the message m part of either the secret or the public component of the zero-knowledge proof (in this case it was added to the public part). However, the witness is Alice's secret key, just as we wanted and the agent Alice did not have to be made public. In fact, the A used inside the zero-knowledge proof function is merely a stand-in which enables the formula F to reduce to true according to the specified equational theory.

One thing is however slightly confusing in the example specified in the paper: the formula F contains the equality "=". This was never defined. We instead suspect that they stated their example to make it more human-readable and that the actual example should read as follows:

$$\mathsf{ZK}(k; m, \mathsf{enc}_{\mathsf{sym}}(m, k); \mathsf{eq}(\beta_2, \mathsf{enc}_{\mathsf{sym}}(\beta_1, \alpha_1)))$$

It should also be pointed out that the authors have introduced equational rules to extract both the public terms N_l as well as the formula F from the function $\mathsf{ZK}_{i,j}$. The former is necessary to model the extraction of the public component. In the previous model from [6] this was modeled as a deduction rule of the adversary. It seems that the latter, i.e. the extraction of the formula F is necessary to pass it as a parameter to the function $\mathsf{Ver}_{i,j}$ where it is required in order to check whether it can be reduced to true.

In many ways we can see that we have a very similar definition as the one presented in [6]: both separate the formula – which again can be understood to encode the predicates of the witness relation – from the actual assignments of M_k to α_k and N_l to β_l respectively. Both models also clearly separate the private from the public component.

The two models handle the content of the formula F however very differently – as we could see when we tried the models on our simple example. While this model also introduces a limited number of function symbols, they are defined as part of a base equational theory. As mentioned above, this equational theory already incorporates many more cryptographic primitives as in [6], but – more importantly – it is clear that this theory can easily be extended by a potential user of the model to include whichever function the user defines. Specifically, the authors only require the formula to consist of the specified function symbols, constants and the terms α_k and β_l. As far as we could determine there were no other restrictions on F.

What is conspicuous in this model compared to the one from [6] is the absence of randomness; both the randomness to construct the proof as well as any randomness required by the functions used inside F. The first means that the prover algorithm P in this case is modeled as a deterministic function. The second would mean that no functions inside the zero-knowledge proof may be probabilistic or they at least have to be modeled as a deterministic function. This can however be circumvented by including randomness as part of the secret component. However, this means that the random values are not clearly separated and this could potentially have its limitations.

The fact that the prover algorithm P however is modeled deterministically is more problematic: with the prover being defined as a probabilistic algorithm (see section 2.2), this does not model non-interactive zero-knowledge proofs correctly. In particular, two proofs of the same statement on the same common reference string would be identical in the deterministic setting, this could violate the security properties of certain protocols.

Even more problematic is that, just as in [6], the common reference string is not part of the model. In the previous model however, the authors at least include the abstract randomness R as part of the zero-knowledge proof function. This means that we could potentially encode the common reference string as the randomness R, however this also does not correctly model non-interactive zero-knowledge proofs.

There is another very important characteristic of this model: the zero-knowledge proof function is defined with arbitrary arity. This means that what they have defined is actually an infite set of equational rules. The reason behind this is that each zero-knowledge proof requires its own equational rules corresponding to its witness relation. The biggest question that arises is then of course how was this implemented. The authors claim to have developed a compiler which takes as input a zero-knowledge proof description and outputs the ProVerif specifications for that proof. As we were not able to obtain the compiler itself despite multiple attempts to contact the authors, we unfortunately can make no further statements regarding its use and how the zero-knowledge proofs have to be encoded.

Backes et al. 2013 computational soundness paper [8]

While the models from [6] and [7] show many similarities, the model from [8] takes a different approach.

In this model a zero-knowledge proof is defined as the function ZK of arity 4. Its parameters are specified to be of the following form: $(\text{crs}(N_1), x, w, N_2)$, where x and w can be any terms and N_1 and N_2 are nonces (which are terms themselves). The function $\text{crs}(N)$ models the common reference string. As is expected, x denotes the public component, i.e. the statement, and w the secret component, i.e. the witness. The nonce N_2 refers to the randomness used to construct the proof (i.e. just as in [6], the prover here is again modeled correctly as a probabilistic algorithm).

There are two major differences to the previously introduced models which jump right out: first, they do introduce the common reference string into their model and second, there is no explicit formula F.

The introduction of the common reference string is not surprising, in fact its absence in the previous two models was far more questionable as this does not correspond to the definition of a non-interactive zero-knowledge proof (see section 2.2). It is however not clear why they chose to leave out the explicit definition of the formula F. In the other models F was crucial to the verification process as it constituted an encoding of the witness relation. Here, the verification process is modeled by the following equation:

$$\begin{aligned} &\text{verify}_{\text{ZK}}(\text{crs}(t_1), \text{ZK}(\text{crs}(t_1), t_2, t_3, t_4)) \\ &\quad = \text{ZK}(\text{crs}(t_1), t_2, t_3, t_4) \text{ , if} \\ &\quad (t_2, t_3) \in R^{\text{sym}}_{\text{adv}} \end{aligned}$$

We will not go into detail of the exact definition of $R^{\text{sym}}_{\text{adv}}$, this relation however can be understood as a symbolic representation of the witness relation. We will look at the meaning of the equation reducing to the function ZK(.) further down, but if we interpret this as denoting that the zero-knowledge proof verifies if first of all the common reference string is the same and second of all if the pair (t_2, t_3), i.e. the statement and the witness, are a member of the witness relation, then this seems to rather closely represent the mathematical definition of a zero-knowledge proof.

In a sense, the relation $R^{\text{sym}}_{\text{adv}}$ plays the role of the formula F from the previous models. The issue in this model however is that the definition of the equation for $\text{verify}_{\text{ZK}}(.)$ does not constitute a well-defined equational theory. The actual verification is deferred to whether or not $(t_2, t_3) \in R^{\text{sym}}_{\text{adv}}$. But how $R^{\text{sym}}_{\text{adv}}$ is encoded and how it can be modeled in an equational theory whether or not (t_2, t_3) is a member of that relation is not explained. Since we were unable to obtain a copy of the source code despite multiple attempts to contact the authors, we could unfortunately also not inspect the code to understand the implementation of this verification process. This would have been a crucial part in understanding the advantages of this particular model and the reason why the formula F was no longer explicitly defined as in the other models.

Just as for the previous models, we would like to give an example of a zero-knowledge proof in this model using our running example we introduced in section 2.2. With the statement x and the witness w being direct parameters of the function ZK(.), using the function symbols defined in [8], the zero-knowledge proof itself is simply:

$$\text{ZK}(\text{crs}(N_1), \text{enc}(\text{ek}(N_2), m, N_3), \text{dk}(N_2), N_4)$$

In order to model the verification process, we also need to define the corresponding relation $R^{\text{sym}}_{\text{adv}}$ using the function symbols of this particular model:

$$R^{\text{sym}}_{\text{adv}} := \{(x, w) \mid \exists N_1, N_2, m.\ x = \text{enc}(\text{ek}(N_1), m, N_2) \wedge w = \text{dk}(N_1)\}$$

The reader might wonder why $\text{crs}(N)$, $\text{ek}(N)$ and $\text{dk}(N)$ are functions taking as input a nonce. While not made explicit in the paper, this refers to the fact that these are constructed using a random value. This in particular means that while the public key $\text{ek}(N)$ and secret key $\text{dk}(N)$ can be linked together, they are not linked by an agent's name and instead the nonce N (just as in the model from [7]).

We want to point out that the model presented above is the closest to the definition of our example we have gotten, i.e. there were no difficulties in translating our example into this model. Indeed, this is an example given by the authors themselves in the paper. The only issue we see is – as we have mentioned – that it is not clear how this translates to an implementation.

We additionally want to highlight some other particularities to this model. Looking at the model in its entirety, it is quite large and the reasons behind some of the authors' decisions are not always comprehensible from the paper alone. We have already seen that the function $\text{verify}_{\text{ZK}}(.)$ reduces to $\text{ZK}(.)$ instead of a symbolic representation of the Boolean constant *true*, as we would expect. This is done similarly in other equations which represent Boolean functions, i.e. they reduce to one of the parameters. There could be advantages to this approach however. For example the reduced term may again be an input to another function. An example for this is given by the authors that the result of $\text{verify}_{\text{ZK}}(.)$ could be an input to the function $\text{getPub}(.)$. The actual reasoning behind this approach and how big of an advantage it truly is versus the confusion it introduces is unfortunately guess work without the source code.

The authors have indeed introduced many such "pseudo-Boolean" functions, such as $\text{isenc}(.)$, $\text{issig}(.)$ or $\text{iszk}(.)$, which model whether or not a term is an encryption, a signature or a zero-knowledge proof respectively. The authors describe these functions as follows: "These are useful for testing properties of terms: The protocol can, e.g., compute $\text{isek}(t)$ and then branch depending on whether the destructor succeeds". However, they do not give an example where this is truly useful and necessary. Without an example, this only seems to make the model unnecessarily complex.

Another reason for the size of the model is that the authors introduce function symbols for various kinds of ill-formed messages, i.e. ill-formed nonces, encryptions, signatures and zero-knowledge proofs, which most likely is used to model incorrect or adversarial protocol use. Just as in [6], these are referred to as "garbage" terms. There exist many equations which handle the reduction of these various types of ill-formed messages, which greatly contributes to the size of the model. It is not clear what the advantage is of this specific handling of ill-formed messages, especially since nothing of the sort was included in the model from [7].

There is an additional difference between this model and the other two: here bitstrings are explicitly modeled through the functions string_0 and string_1. These construct a string similarly to how the Peano arithmetic can be used to represent numbers. If we use ϵ to represent the empty string then $\text{string}_0(\text{string}_1(\epsilon))$ then represents the bitstring "10". The function $\text{unstring}(.)$ then is used to deconstruct the string. As with the other differences we have pointed out, it is not clear why this was modeled here and is not present in the other models.

As a last remark, we want to note that just like in the other models, the equational theory which can appear in the model is fixed. Even though this is not explicitly stated, it is safe to assume that just as for [7] this can be extended by the user with additional function symbols. Since in the paper the emphasis was however to prove computational soundness for this model, it is not clear if their statements still hold for a user-extended model.

3.2 Standardising the equational theories

As discussed in section 3.1, all existing models have employed various different function symbols, not only to model the zero-knowledge proof itself but all other protocol parts. In this section we therefore want to provide a clear and direct comparison between the different models. In other words, we standardised the various models into a comparable form.

Methodology and organisation

We have collected all constructors and destructors from the papers [6], [7] and [8] and present them in the tables 1 - 9 using the Tamarin terminology and semantics of *terms*, *functions* (i.e. function symbols) and *equations*.

All tables are organised in the same way, where the first column holds a brief description of the function or of what the equation represents. Each following column holds the corresponding term or equation from one of the papers. As a convention, we abbreviate the Backes et al. 2008 paper on computational soundness of zero-knowledge proofs [6] as ***2008-sound-symb***, the Backes et al. 2008 paper on zero-knowledge proofs in the applied pi-calculus [7] as ***2008-pi-calc*** and the Backes et al. 2013 paper on computational soundness as ***2013-sound-symb***. By organising the functions and equations in such a way, we can easily observe all the differences in the models at a glance.

Before noting the most crucial observations that can be drawn from this side-by-side comparison, we want to remark on the following convention: In case a function simply does not exist, i.e. has not been modeled in any way, we mark the cell with a "–". We mark a function symbol to be "not specified" if it most likely exists in some form or another in the model but it wasn't explicitly declared. For example, there exists the clear notion of an ill-formed zero-knowledge proof in [8] (namely $\mathsf{garbageZK}(t, t, N)$), but this is not specified explicitly in [6]. However, more than likely this exists in one form or another.

Additionally, we have used the symbol "**" in some of the equations tables. As we have discussed in section 3.1, *2008-sound-symb* (i.e. [6]) introduces adversary deduction rules instead of term equations in their model. Whenever a deduction rule exists which corresponds to the term equations from the other papers, we have marked this with "**". It must be noted however, that this means that these functionalities only exist for the adversary and cannot be used inside the protocol. The adversary deduction rules and their meaning are discussed further in section 3.3.

We also want to mention that while we have respected the authors' notations to a certain degree, we have made some modifications in order to allow for a better comparison. We also again use our own notational conventions we presented in section 1 whenever possible. We also want to note the following additional conventions:

1. If a function's parameters are explicitly listed, N refers to a nonce parameter, A an agent, S a string, R randomness, F a Boolean formula and t any term.
2. In case the parameters are explicitly specified for a term function, we list them accordingly. In case no parameters are specified, we instead provide the arity of the term function according to Tamarin terminology. For example, if the function term $\mathsf{enc}_{\mathsf{asym}}$ has arity 3, this is denoted as $\mathsf{enc}_{\mathsf{asym}}/3$.

   Since the arities of the functions used to model zero-knowledge proofs are particularly interesting, we anotate their arities explicitly even if the parameters are provided.
3. In case the models make a distinction between values which have been generated by honest protocol participants and the ones which have been generated by the adversary, we denote this as $N = N_{\text{adv}} \cup N_{\text{honest}}$, for some set of names N regardless of the naming conventions used inside the originating paper.

Results

The resulting tables (tables 1 - 9) can be found towards the end of this section.

Observations

We can now draw some conclusions from this side-by-side comparison. First, we can see it is glaringly obvious that there are vast differences between how not only the zero-knowledge proofs, but also how the general and cryptographic functionalities were modeled. Table 2, which lists the term functions for general cryptographic primitives, in particular shows glaring holes. It is not surprising that only *2008-pi-calc* (i.e. [7]) models blind signing, as this is a less used primitive, but which is necessary for the modeling of their case study. However, the holes regarding symmetric encryption and hashing and in the case of *2008-sound-symb* any form of signing at all, are much more surprising and rather inexplicable.

As we can see in table 5, logical operations are only modeled by [7]. In this model, the functions $\wedge$, $\vee$ and eq, as well as the symbolic representations of the Boolean constant true, are vital to model the formula F. In particular, true is essential for the verification process of the zero-knowledge proof. What is not clear is why the constant false was also modeled, as no equational rules were specified which reduce to false. Additionally, the function symbol for true was actually specified twice, once in a base signature Σ_{base} and a second time inside the signature specific to zero-knowledge proofs Σ_{ZK} which is defined as the union of Σ_{base} with some additional symbols, which included true again. While this is not an issue, as the two signatures do not have to be distinct, it is quite confusing. If the authors wanted to highlight that true was essential for zero-knowledge proofs, then this should also have included the Boolean operations we mentioned above.

With regards to the definition of F from *2008-sound-symb* (i.e. [6]) it is clear why these operators did not have to be modeled. And as we can see, indeed no Boolean operations are included in the model at all. Interestingly enough though, the opposite is true for the model *2013-sound-symb* (i.e. [8]), where we strongly suspect that Boolean operations would be necessary to model the witness relation $\mathcal{R}$. As discussed in the section 3.1 however, we do not know how this was implemented. This would have been vital to properly compare the verification process of these three models.

It is also interesting to take a closer look at the general model terms listed in table 1. *2008-sound-symb* (i.e. [6]) is the only model which explicitly models randomness. In *2013-sound-symb* randomness is modeled using nonces, as can be seen in their definitions of the encryption key $\mathsf{ek}(N)$ and their encryption term function $\mathsf{enc}(\mathsf{ek}(N), t, N)$ for example (see table 2). Why the *2008-sound-symb* model makes such a clear distinction between randomness and nonces is not clear. Even more interesting is that *2008-pi-calc* actually does not model randomness in any way, i.e. there is no notion of nonces explicitly mentioned nor do any of the functions include randomness (see for example the equation for asymmetric decryption in table 7). We do however believe that nonces have to be modeled in one way or another since they are mentioned inside their case study.

Two of the models, namely *2008-sound-symb* and *2013-sound-symb* (i.e. [6] and [8] respectively) additionally distinguish between randomness created by the adversary and those created honestly inside a protocol run. We have highlighted this distinction in the table 1 by defining for example the set of nonces Nonce into $\mathsf{Nonce}_{\mathrm{adv}}$ and $\mathsf{Nonce}_{\mathrm{honest}}$. In this definition the sets $\mathsf{Nonce}_{\mathrm{adv}}$ and $\mathsf{Nonce}_{\mathrm{honest}}$ are understood to be distinct.

We also briefly want to discuss the specific modeling of an agent in *2008-sound-symb* (i.e. [6]). On one hand, as we have previously mentioned, this specific inclusion of an agent rather restricts the model as an encryption key has to be bound to an agent, which is considered to be a public value (at least inside zero-knowledge proofs). On the other hand, we suspect that the notion of an agent had to specifically included in this model, since it was not actually implemented. Without employing a tool and with it a specific definition of a protocol and protocol agent, the authors had to provide these definitions themselves.

While not made explicit, it is clear that the models from [6] and [8] use some kind of sort for their messages. For example, [6] definitely distinguishes the sort Rand, Nonce, $\mathsf{Garbage}$ and A to denote agents. In [8] there is not as much of a distinction, mostly because there is no differentiation between randomness and nonces and "garbage" is not considered to be a sort in itself. Instead, this model only distinguishes between the type S which denotes strings and N for nonces. Why there is no mention of any such distinction in the model from [7] is not clear.

Another point worth mentioning is that only *2013-sound-symb* models strings. Why it was included in that model specifically and none of the others or any at all for that matter is not clear.

Lastly, we want to take a closer look at the zero-knowledge proof terms and equations presented in tables 3 and 8 respectively. The function for the zero-knowledge proof itself is of particular interest. First of all, it is not clear what arity the function in the model *2008-sound-symb* has, which is why we marked it with a question mark (see table 3). The parameters Rand and F are introduced rather sneakily as qualifiers to the function, even though they clearly are parameters to the function itself. This means that the function has at least arity 5. However, with regards to the three tuple parameters inside the brackets it is not clear how these tuples of variable length are implemented. The question is whether each tuple parameter is to be understood as a single parameter or, as in the model *2008-pi-calc*, as multiple parameters. The latter would then also lead to a variable arity of the $\mathsf{ZK}_F^{\mathsf{Rand}}$ function, just as in *2008-pi-calc*.

What is even more interesting however, is the function ZK in the model *2013-sound-symb*. The arity here is fixed and the private and public components of the zero-knowledge proof are simply modeled as singular terms. With a recursive use of the pair function however we could easily build a sort of tuple with variable length which can then be passed to this function. This was not mentioned explicitly.

What is clear however, is that both the zero-knowledge proof function and the corresponding verification is not straight-forward. **None** of the verification "equations" presented in table 8 are a simple term equation as is the case for the other cryptographic primitives modeled (see table 7 for details). And while the models seemed to almost want to hide this fact, the verification functions in particular are specific to the zero-knowledge proof being modeled and are not generic. It could be interesting, to investigate further what the advantages of the verification process in the model *2008-sound-symb* could be and whether the idea of "true zero-knowledge proofs" could lead to a generic implementation. It is however conspicuous that in the other two models – which were indeed implemented – this approach was abandoned.

3.3 Additional adversary deduction rules

Both ProVerif and Tamarin come with built-in deduction rules, modeling a Dolev-Yao adversary. Two of the models however, namely the ones presented in [6] and [8], introduce additional deduction rules for the adversary only. In case of the former, no term equations were even defined. Instead, the only rules the adversary may apply are the specifically defined deduction rules.

We do want to point out a clear distinction between equational rules and adversary's deduction rules. The former may be used by any (honest or adversarial) party in the protocol. The latter however, may only be used by an adversary. It is clear that we do want to model the adversary as strong as possible in order to make meaningful proofs. However, it is rather strange to model something as simple as a message decryption for the adversary only – which was indeed done in [6].

We will now present these (additional) adversary deduction rules given in these two models. Note that we use the notation we have presented in the tables in section 3.2.

Deduction rules in the Backes et al. 2008 computational soundness paper [6]

The adversary deduction rules can be found in figure 1 in [6]. Most of them are quite straight-forward. We nonetheless, want to give a full list here for completeness purposes.

The authors introduce what they call a deduction relation $\vdash$, whereas they denote by $\varphi \vdash t$ that the adversary can deduce the term t from φ. What exactly φ stands for is not explained in the paper, however it most likely refers to the adversary's knowledge which includes all messages which have been sent. $\varphi \vdash t$ can also be understood as the adversary can construct the term t from his knowledge. The rules are presented in the style $\frac{\text{premise}}{\text{conclusion}}$.

Below we list all the deduction rules and translate their meaning to plain English:

1. $\frac{m \in \varphi}{\varphi \vdash m}$: If the term m is known to the adversary, he can deduce m.

2. $\frac{g, g' \in \mathsf{Garbage} r \in \mathsf{Rand}_{\text{adv}}}{\varphi \vdash g \;\; \varphi \vdash \mathsf{ek}(g) \;\; \varphi \vdash \{g'\}^r_{\mathsf{ek}(g)}}$: The adversary can create a ciphertext using any "garbage" terms and its own randomness.

3. $\frac{b \in \mathsf{A}}{\varphi \vdash \mathsf{ek}(b) \;\; \varphi \vdash b}$: For any agent b, the adversary knows b and can deduce his public (encryption) key $\mathsf{ek}(b)$.

4. $\frac{\varphi \vdash m_1 \;\; \varphi \vdash m_2}{\varphi \vdash \langle m_1, m_2 \rangle}$: If the adversary knows the terms m_1 and m_2, then he can construct their pairing $\langle m_1, m_2 \rangle$.

5. $\frac{\varphi \vdash \langle m_1, m_2 \rangle}{\varphi \vdash m_1 \;\; \varphi \vdash m_2}$: If the adversary knows the pairing of two terms, he can deduce each individual term. This corresponds to the "first" and "second" functions from the other models (see table 6 in section 3.2 for details).

6. $\frac{\varphi \vdash \mathsf{ek}(b) \;\; \varphi \vdash m \;\; r \in \mathsf{Rand}_{\text{adv}}}{\varphi \vdash \{m\}^r_{\mathsf{ek}(b)}}$: If the adversary knows a message m, he can encrypt m with the public key of an agent b using its own randomness.

7. $\frac{\varphi \vdash \{m\}^r_{\mathsf{ek}(b)} \quad \varphi \vdash \mathsf{dk}(b)}{\varphi \vdash m}$: If the adversary has compromised an agent b, i.e. knows his private (decryption) key $\mathsf{dk}(b)$, the adversary can deduce the plain text message corresponding to a ciphertext encrypted with b's public (encryption) key.

8. $\frac{\varphi \vdash \mathsf{ZK}^r_F(\underline{r};\underline{a};\underline{b})}{\varphi \vdash \underline{b}}$: If the adversary has received a zero-knowledge proof, he can extract the public component.

9. If the adversary knows both the statement and the witness which satisfy the formula F, he can create a valid zero-knowledge proof, wherein he may use his own randomness to construct the proof and may reuse some randomness extracted from honest ciphertexts for the random values "inside" the proof.[15]

There are some important observations to be made regarding the handling of randomness in the last deduction rule. As mentioned above there are two "types" of randomness involved in this rule. One is the randomness to construct the proof itself and the other are random values used inside the formula F. The authors state that the former has to be a random value created by the adversary, whereas the other random values can either have been created by the adversary or extracted from a ciphertext which has been encrypted with an honest agent's encryption key, but whose decryption key has been compromised[16].

While the latter part makes the rule rather complicated it is necessary to ensure computational soundness. The authors state that if instead they would only allow the adversary to use its own randomness, then some protocols appear to be secure even though they aren't. As an example they give a zero-knowledge proof which involves a ciphertext, which was encrypted using a random value r_i. If the adversary is required to use his own randomness $r_j \in \mathsf{Rand}_{\mathrm{adv}}$ in the proof, then the proof would not verify, i.e. the protocol would appear secure, since $r_i \neq r_j$. But in the real world, if the adversary knows r_i then he could construct a valid proof. Hence, they allow for this situation.

The authors therefore include the adversary's capability to extract the randomness used in an encryption through the following premise inside this last deduction rule: $\forall i. r_i \in \mathsf{Rand}_{\mathrm{adv}} \vee (\exists t, a. \varphi \vdash \{t\}^{r_i}_{\mathsf{ek}(a)} \wedge \varphi \vdash \mathsf{dk}(a))$, whereas the r_i denote the values inside the tuple $\underline{r}$, which will be used for the construction of the zero-knowledge proof.

This last deduction rule is particularly interesting, because the authors have stressed the importance of where the randomness comes from inside the proofs, however none of the other papers – which chronologically appeared later – even separately model the random values. Random values may be part of the secret

[15]Given the length of this rule and the fact that it includes many notations which we have not introduced in this thesis, we have opted not to re-print the rule itself. For the formal definition of this rule we refer the reader to figure 1 in the original paper [6].

[16]As mentioned in [6] even if a cryptosystem is IND-CCA secure, this does not imply that the randomness cannot be retrieved provided one can decrypt the message (i.e. the decryption key is compromised).

component, i.e. the witness, but they don't have to be. This seems to be particularly interesting, since this particular model seems to allow us to model cases where the adversary indeed reuses randomness. The question arises however then why this was no longer deemed necessary later on. Perhaps the reason is that this was not of such significance as the authors first claimed and indeed the more crucial question arises regarding the randomness used to construct the proof itself. None of the papers however, delve further into this question.

Deduction rules in the Backes et al. 2013 computational soundness paper [8]

The adversary's deduction rules can be found in chapter 2. Similarly to [6], they introduce the deduction relation $\vdash$. Instead of φ they use a set of terms S. They say that $S \vdash t$ means that from the terms S, the adversary can deduce t. Again, it is assumed that S denotes the adversary's knowledge, i.e. the set of terms the adversary knows. The rules are presented, as before, in the style $\frac{\text{premise}}{\text{conclusion}}$:

1. $\frac{m \in S}{S \vdash m}$: If m is known to the adversary, he can deduce m.

2. $\frac{N \in N_{\text{adv}}}{S \vdash N}$: We recall that the set N_{adv} refers to the set of nonces created by the adversary. This rule therefore states that the adversary can deduce any adversarial nonces.

3. $\frac{S \vdash t_1, ..., t_n \;\; t_1, ..., t_n \in \mathbf{T} \;\; \mathsf{F}(t_1, ..., t_n) \in \mathbf{T}}{S \vdash \mathsf{F}(t_1, ..., t_n)}$: $\mathbf{T}$ is the set of all terms and F a function symbol. The rule therefore states that if the adversary can deduce the terms $t_1, ..., t_n$, then he can also deduce $\mathsf{F}(t_1, ..., t_n)$.

These deduction rules are quite straight-forward and represent what one might expect with regards to the adversary's capabilities (see for example the message deduction rules defined for Tamarin in [11]). What is conspicuous however, is that no additional adversary capability was modeled. In Tamarin for example it is good practice to model the ability of the adversary to compromise any agent through additional protocol-specific rules. With a clear definition of secret decryption and signing keys, it is not clear why the authors decided not to model this explicitly.

There is another conspicuous absence in these deduction rules. If we look at the rules given in [11] we can see that any message sent out and any received is communicated through the adversary (see section 2.1 for the definition of the Dolev-Yao adversary and its specificiation inside Tamarin). This particular behaviour is not modeled here. Instead the authors state that this capability of intercepting and modifying messages has to be modeled explicitly by the protocol, which explains its absence in the above deduction rules. However, it is not clear – nor explained in the paper – why it was not simply included.

3.4 Concluding remarks

As we have seen, the three models have vastly different approaches with regards to the modeling of zero-knowledge proofs, and in fact modeling of protocols in general. Since we have discussed the most significant differences already in detail in the previous sections, we would like to now focus on any potential advantages or disadvantages one approach might have over the other.

We first have to point out that the missing common reference string in the models from [6] and [7] is a clear issue as it in fact violates the definition of a non-interactive zero-knowledge proof. One could argue that if we assume the common reference string to have been honestly created that it can be abstracted away in the symbolic model. However, it is clear that there are real-life attacks which are possible if the common reference string is not honestly generated (see section 2.2). Without the inclusion of the common reference string, we cannot model these attacks. We also want to point out that none of these limitations on the model were discussed in [6].

Also a clear issue is the fact that the zero-knowledge proof function in [7] was modeled deterministically instead of as a probabilistic algorithm. While some provers may of course run deterministically, this does not capture the general behaviour and properties of non-interactive zero-knowledge proofs[17] and, as previously mentioned, this is again not in accordance with their mathematical definition (see section 2.2). This is all the more strange, since the inclusion of randomness in a function is very simple in the symbolic model.

Unsurprisingly, the analysis of the existing models has shown that the crux in designing a model for zero-knowledge proofs is the representation of the proof verification process and with it the encoding of the witness relation. Whether or not a proof is accepted as valid is highly specific to each zero-knowledge proof and it therefore is no surprise that a generic verification process is not as simple as the verification of a signature for example which always has the same criteria. Unfortunately, the solutions proposed by these models are not satisfying. This could be because, as already mentioned, we were unable to obtain a copy of the source code of any of the models despite multiple contact attempts. Perhaps, by analysing the implementation alongside the papers, the encoding of the witness relation would have become clearer and we could see the strong points of each approach. As it is now, we must come to the following conclusions.

With the notion of "true zero-knowledge proofs", the model from [6] has a very promising approach. Without an equational theory, they circumvent the issue of having to deal with an infinite set of equational rules for a generic model. However, the proposed model is much too restrictive – as has been exemplified by our attempt to model our very simple running example. As far as we know there exists no implementation of this model. It would therefore be interesting to attempt such an implementation and see whether the model

[17]In certain protocols it could potentially be vital that two zero-knowledge proofs making the same statement and created with the same common reference string are distinct.

could be extended to include other cryptographic primitives. We leave that as an open question.

The biggest issue for the model of [7] is that their model actually requires an infinte set of equational rules. We recall that the authors describe a compiler which translates a zero-knowledge proof specification into a proper ProVerif model with a finite set of equational rules particular to that zero-knowledge proof. This seems like a very interesting and elegant way for dealing with this issue. The problem here lies however in the definition of the formula F. This formula is constructed using user-defined function symbols to represent Boolean operators and is in that sense not a "proper" logical formula. The authors state that ProVerif would no longer terminate in some cases when provided with the output of their compiler. This was apparently due to the presence of the function symbols $\wedge$ and $\vee$ and their corresponding equations. They state a theorem by which the equational theory output by the compiler can be modified to no longer contain these function symbols. However, they do not guarantee that it can always be applied and merely state that this is often the case.

In many ways the model from [8] is the most promising. This is due to its closeness to the mathematical definition of non-interactive zero-knowledge proofs (see section 2.2). However, we have no idea how the witness relation is encoded nor how the check of a membership in the witness relation can be done in an equational theory. As mentioned above, this missing information relates exactly to the crux of a zero-knowledge proof model and therefore is the most interesting part. This makes the fact that we could not obtain the source code for this particular model even more disappointing.

An advantage which is present – at least in theory – in all three models is that they explicitly and clearly encode or even directly define the witness relation. A more expressive model is by its nature more user-friendly and thus reduces the risk of human-error in the implementation of a protocol. Depending on the actual implementation of the model, this advantage could be lost again however.

Given the extensiveness of some of these models, e.g. by modeling "pseudo-Boolean" functions and having various terms to represent ill-formed messages, the reader might naturally wonder if a zero-knowledge proof model in general requires large models. As we will see in our own models proposed in sections 4 and 6.1, this is not the case. We were able to keep our user-defined model quite slim especially by using some – but not many – Tamarin built-in functionalities and already built-in adversary deduction rules. We suspect that in particular the model in [8] ended up quite large as they were trying to take advantage of some inner workings of the CoSP framework. We therefore want to stress that some of the differences of the model specifications might be due to the differences in the tools and/or frameworks used in the background.

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Agent	A	(not specified)	(not specified)
Nonce	$\mathsf{Nonce} = \mathsf{Nonce}_{\text{honest}} \cup \mathsf{Nonce}_{\text{adv}}$	(not specified)	$N = N_{\text{honest}} \cup N_{\text{adv}}$
Randomness	$\mathsf{Rand} = \mathsf{Rand}_{\text{honest}} \cup \mathsf{Rand}_{\text{adv}}$	–	–
String	–	–	S
Empty string	–	–	empty
Adding “0” to a bitstring S	–	–	$\mathsf{string}_0(S)$
Adding “1” to a bitstring S	–	–	$\mathsf{string}_1(S)$
Removing “0” from a bitstring S	–	–	$\mathsf{unstring}_0(S)$
Removing “1” from a bitstring S	–	–	$\mathsf{unstring}_1(S)$
Pairing of terms	$\langle t, t \rangle$	$\mathsf{pair}/2$	$\mathsf{pair}(t, t)$
First term of pair	–	$\mathsf{first}/1$	$\mathsf{fst}/1$
Second term of pair	–	$\mathsf{snd}/1$	$\mathsf{snd}/1$

Table 1: Standardised form of general model **function symbols** in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Hash	–	$\mathsf{hash}/1$	–
Encryption (public) key	$\mathsf{ek}(A)$	$\mathsf{pk}/1$	$\mathsf{ek}(N)$
Decryption (private) key	$\mathsf{dk}(A)$	$\mathsf{sk}/1$	$\mathsf{dk}(N)$
Secret (signing) key	–	$\mathsf{sk}/1$	$\mathsf{sk}(N)$
Verification key	–	$\mathsf{pk}/1$	$\mathsf{vk}(N)$
Symmetric encryption	–	$\mathsf{enc}_{\mathsf{sym}}/2$	–
Symmetric decryption	–	$\mathsf{dec}_{\mathsf{sym}}/2$	–
Asymmetric encryption	$\{t\}^{R}_{\mathsf{ek}(A)}$	$\mathsf{enc}_{\mathsf{asym}}/2$	$\mathsf{enc}(\mathsf{ek}(N), t, N)$
Asymmetric decryption	–	$\mathsf{dec}_{\mathsf{asym}}/2$	$\mathsf{dec}/2$
Key of an encryption	–	–	$\mathsf{ekof}/1$
Verification key of a signature	–	–	$\mathsf{vkof}/1$
Signing	–	$\mathsf{sign}/2$	$\mathsf{sig}(\mathsf{sk}(N), t, N)$
Signature verification	–	$\mathsf{ver}/3$	–
Message of a signature	–	$\mathsf{msg}/1$	–
Blinding	–	$\mathsf{blind}/2$	–
Unblinding	–	$\mathsf{unblind}/2$	–
Blind signing	–	$\mathsf{blindsign}/2$	–
Blind signature verification	–	$\mathsf{blindver}/3$	–
Message of a blinded signature	–	$\mathsf{blindmsg}/1$	–

Table 2: Standardised form of **function symbols** for general cryptographic primitives in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Common reference string	–	–	$\mathsf{crs}(N)$
Zero-knowledge proof	$\mathsf{ZK}^R_F(\underline{R};\underline{t};\underline{t})/?$	$\mathsf{ZK}_{i,j}(\underline{t},\underline{t},F)/(i+j+1)$	$\mathsf{ZK}(\mathsf{crs}(N),t,t,N)/4$
Verification of a zero-knowledge proof	– (instead a definition of "true zero-knowledge proofs" is given, see definition 2 in [6])	$\mathsf{Ver}_{i,j}/2$	$\mathsf{verify}_{\mathsf{ZK}}/2$
Public component of a zero-knowledge proof	–	$\mathsf{Public}_i/1$	$\mathsf{getPub}/1$
Zero-knowledge proof formula	(not specified)	$\mathsf{Formula}/1$	–
Common reference string of a zero-knowledge proof	–	–	$\mathsf{crsof}/1$

Table 3: Standardised form of zero-knowledge proof **function symbols** of existing zero-knowledge proof models. The arity of the zero-knowledge proof function in column *2008-sound-symb* is marked with a question mark, since its arity is not well-defined.

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Ill-formed nonce	$\mathsf{Garbage}$	–	$\mathsf{garbage}(N)$
Ill-formed encryption key	$\mathsf{ek}(\mathsf{Garbage})$	–	(not specified)
Ill-formed signature	–	–	$\mathsf{garbageSig}(t,N)$
Ill-formed ciphertext	$\mathsf{enc}(\mathsf{ek}(\mathsf{Garbage}),\mathsf{Garbage},R)$	–	$\mathsf{garbageEnc}(t,N)$
Ill-formed zero-knowledge proof	(not specified)	–	$\mathsf{garbageZK}(t,t,N)$

Table 4: Standardised form of **function symbols** representing ill-formed messages in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Logical and	–	$\wedge/2$	–
Logical or	–	$\vee/2$	–
Equality of terms	–	eq/2	equals/2
Boolean value “true”	–	true/0	–
Boolean value “false”	–	false/0	–
Type query: encryption key	–	–	isek
Type query: verification key	–	–	isvk
Type query: encryption	–	–	isenc
Type query: signature	–	–	issig
Type query: common reference string	–	–	iscrs
Type query: zero-knowledge proof	–	–	isZK

Table 5: Standardised form of **function symbols** for auxiliary Boolean operations and functions in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Retrieval of first term of pair	**	$\mathsf{first}(\mathsf{pair}(t_1, t_2)) = t_1$	$\mathsf{fst}(\mathsf{pair}(t_1, t_2)) = t_1$
Retrieval of second term of pair	**	$\mathsf{snd}(\mathsf{pair}(t_1, t_2)) = t_2$	$\mathsf{snd}(\mathsf{pair}(t_1, t_2)) = t_2$
Deconstructing a "0"-extended string	–	–	$\mathsf{unstring}_0(\mathsf{string}_0(s)) = s$
Deconstructing a "1"-extended string	–	–	$\mathsf{unstring}_1(\mathsf{string}_1(s)) = s$

Table 6: Standardised form of general **equations** in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Signature verification	–	$\mathsf{ver}(\mathsf{sign}(x, \mathsf{sk}(y)), x, \mathsf{pk}(y)) = \mathsf{true}$	$\mathsf{verify}_{\mathsf{sig}}(\mathsf{vk}(t_1), \mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)) = t_2$
Symmetric decryption	–	$\mathsf{dec}_{\mathsf{sym}}(\mathsf{enc}_{\mathsf{sym}}(x, y), y) = x$	–
Asymmetric decryption	**	$\mathsf{dec}_{\mathsf{asym}}(\mathsf{enc}_{\mathsf{asym}}(x, \mathsf{pk}(y)), \mathsf{sk}(y)) = x$	$\mathsf{dec}(\mathsf{dk}(t_1), \mathsf{enc}(\mathsf{ek}(t_1), m, t_2)) = m$
Retrieval of message part of a signature	–	$\mathsf{msg}(\mathsf{sign}(x, y)) = x$	–
Blind signature verification	–	$\mathsf{blindver}(\mathsf{unblind}(\mathsf{blindsign}(\mathsf{blind}(x, z), \mathsf{sk}(y)), z), x, \mathsf{pk}(y)) = \mathsf{true}$	–
Retrieval of message part of a blind signature	–	$\mathsf{blindmsg}(\mathsf{unblind}(\mathsf{blindsign}(\mathsf{blind}(x, z), y), z)) = x$	–
Retrieval of key of an encryption	–	–	$\mathsf{ekof}(\mathsf{enc}(\mathsf{ek}(t_1), t_2, t_3)) = \mathsf{ek}(t_1)$
Retrieval of key of an ill-formed encryption	–	–	$\mathsf{ekof}(\mathsf{garbageEnc}(t_1, t_2)) = t_1$
Retrieval of verification key of a signature	–	–	$\mathsf{vkof}(\mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)) = \mathsf{vk}(t_1)$
Retrieval of verification key of an ill-formed signature	–	–	$\mathsf{vkof}(\mathsf{garbageSig}(t_1, t_2)) = t_1$

Table 7: Standardised form of **equations** of general cryptographic primitives in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Zero-knowledge proof verification	– (instead a definition of “true zero-knowledge proofs” is given, see definition 2 in [6])	$\mathsf{Ver}_{i,j}(F, \mathsf{ZK}(\underline{M}; \underline{N}; F)) = \mathsf{true}$ iff condition detailed in section 3.1 holds	$\mathsf{verify}_{\mathsf{ZK}}(\mathsf{crs}(t_1), \mathsf{ZK}(\mathsf{crs}(t_1), t_2, t_3, t_4)) = \mathsf{ZK}(\mathsf{crs}(t_1), t_2, t_3, t_4)$, if $(t_2, t_3) \in \mathcal{R}$
Retrieval of public component	**	$\mathsf{Public}_p(\mathsf{ZK}(\underline{M}; \underline{N}; F)) = N_p$	$\mathsf{getPub}(\mathsf{ZK}(t_1, t_2, t_3, t_4)) = t_2$
Retrieval of public component of ill-formed zero-knowledge proof	–	–	$\mathsf{getPub}(\mathsf{garbageZK}(t_1, t_2, t_3)) = t_2$
Retrieval of common reference string of a zero-knowledge proof	–	–	$\mathsf{crsof}(\mathsf{ZK}(\mathsf{crs}(t_1), t_2, t_3, t_4)) = \mathsf{crs}(t_1)$
Retrieval of common reference string of an ill-formed zero-knowledge proof	–	–	$\mathsf{crsof}(\mathsf{garbageZK}(t_1, t_2, t_3)) = t_1$

Table 8: Standardised form of zero-knowledge proof **equations** in existing zero-knowledge proof models

Description	**2008-sound-symb** [6]	**2008-pi-calc** [7]	**2013-sound-symb** [8]
Equality of terms	–	$\mathsf{eq}(x, x) = \mathsf{true}$	$\mathsf{equals}(x, x) = x$
Logical and	–	$\wedge(\mathsf{true}, \mathsf{true}) = \mathsf{true}$	–
Logical or (1)	–	$\vee(x, \mathsf{true}) = \mathsf{true}$	–
Logical or (2)	–	$\vee(\mathsf{true}, x) = \mathsf{true}$	–
Type query: encryption key	–	–	$\mathsf{isek}(\mathsf{ek}(t)) = \mathsf{ek}(t)$
Type query: verification key	–	–	$\mathsf{isvk}(\mathsf{vk}(t)) = \mathsf{vk}(t)$
Type query: encryption	–	–	$\mathsf{isenc}(\mathsf{enc}(\mathsf{ek}(t_1), t_2, t_3))$ $= \mathsf{enc}(\mathsf{ek}(t_1), t_2, t_3)$
Type query: ill-formed encryption	–	–	$\mathsf{isenc}(\mathsf{garbageEnc}(t_1, t_2))$ $= \mathsf{garbageEnc}(t_1, t_2)$
Type query: signature	–	–	$\mathsf{issig}(\mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3))$ $= \mathsf{sig}(\mathsf{sk}(t_1), t_2, t_3)$
Type query: ill-formed signature	–	–	$\mathsf{issig}(\mathsf{garbageSig}(t_1, t_2))$ $= \mathsf{garbageSig}(t_1, t_2)$
Type query: common reference string	–	–	$\mathsf{iscrs}(\mathsf{crs}(t_1) = \mathsf{crs}(t_1)$
Type query: zero-knowledge proof	–	–	$\mathsf{isZK}(\mathsf{ZK}(t_1, t_2, t_3, t_4))$ $= \mathsf{ZK}(t_1, t_2, t_3, t_4)$
Type query: ill-formed zero-knowledge proof	–	–	$\mathsf{isZK}(\mathsf{garbageZK}(t_1, t_2, t_3))$ $= \mathsf{garbageZK}(t_1, t_2, t_3)$

Table 9: Standardised form of **equations** representing auxiliary Boolean operations and functions in existing zero-knowledge proof models

4 Non-interactive zero-knowledge proofs in Tamarin

Having analysed the existing ProVerif and CoSP models in section 3, we now propose a general zero-knowledge proof model for Tamarin. We first outline what it is we want our model to achieve and any challenges and pitfalls we expect before introducing the model itself. Just as with the existing models in section 3, we then apply our model to the simple example we introduced in the background section 2.2. We will then further test it on some additional zero-knowledge proofs.

For any questions regarding the Tamarin prover and its syntax, we refer the reader to the section 2.1.3 or [15], [4].

4.1 From the cryptographic definition to a symbolic model

The main goal of our model is to provide the user with a simple either built-in or easy to declare cryptographic primitive which can be incorporated into security protocols containing zero-knowledge proofs. Ideally, this should be as simple as modeling a hash or an encryption function. We have opted not to add a built-in zero-knowledge proof functionality to Tamarin. Instead we present generic explanations and instructions how a user may easily implement a zero-knowledge proof himself with hardly any overhead.

The reason behind this is similar to the challenges faced by the existing models we discussed in section 3. A model is dependent on the actual zero-knowledge proof itself. This means we have to deal with the fact that a statement and/or witness may consist of multiple terms and have to somehow find a way to model the verifier's behaviour according to the definition of the witness relation (see section 2.2 for details). As the authors of [7] have mentioned, this actually leads to an infinite number of equational rules, which obviously is not possible to implement.[18].

According to the definition of a zero-knowledge proof we have given in section 2.2, we can identify the following components which need to be modeled:

- The **probabilistic algorithms** P and V
- The **zero-knowledge proof**, i.e. the message which will be constructed by the prover and sent to the verifier[19]
- The **statement**, i.e. the public component, of the zero-knowledge proof and the **witness**, i.e. the secret component

[18]In order to avoid this issue, the authors have developed a compiler which encodes a zero-knowledge proof description into ProVerif specifications (see section 3). As we will see later on – especially in section 6.1 – we have found an elegant way to model zero-knowledge proofs, which requires such minimal implementation on the user's side, that there is no need for such a compiler.

[19]As discussed in section 2.2, a zero-knowledge proof in the non-interactive setting can be considered as a single message.

Function symbols

$$\begin{array}{l}\mathsf{zkp}(\mathsf{crs}, \underline{x}, \underline{w}, r)\\ \mathsf{verifyZkp}(\mathsf{crs}, \mathsf{zkp}(.))\\ \mathsf{trueZkp}\\ \mathsf{pubParams}_i(\mathsf{zkp}(.))\end{array}$$

Equations

$$\begin{array}{l}\mathsf{verifyZkp}(\mathsf{crs}, \mathsf{zkp}(\mathsf{crs}, \underline{x}, \underline{w}, r)) = \mathsf{trueZkp}, \text{ if } (\underline{x}, \underline{w}) \in \mathcal{R}\\ \mathsf{pubParams}_i(\mathsf{zkp}(\mathsf{crs}, \underline{x}, \underline{w}, r)) = x_i\end{array}$$

Figure 1: Equational theory for zero-knowledge proofs. The encoding of the condition "if $(\underline{x}, \underline{w}) \in \mathcal{R}$" is done by potentially declaring multiple equational rules for the function symbol verifyZkp.

- Since anyone may extract the public statement from a zero-knowledge proof, we need a way to model such an "**extractor**"
- The generation of the **common reference string**
- A suitable encoding of the **witness relation** $\mathcal{R}$
- The **verification process** of a zero-knowledge proof, i.e. the check whether the statement/witness pair $(x, w) \in \mathcal{R}$
- The potential **reveal of a witness** if an agent is compromised

We model the above components through an equational theory we present in figure 1 and several user-defined infrastructure as well as adversary rules. We will now introduce this model in detail.

Modeling the prover and the zero-knowledge proof

In the formal definition of a zero-knowledge proof from section 2.2 we can see that the prover P is defined as a probabilistic algorithm $P(x, CRS)$. In other words, the only input parameters to this algorithm are the statement x and the common reference string.

The prover P itself can easily be modeled as a protocol agent, its functionality however can be understood as the zero-knowledge proof itself, since this is precisely what is then sent to a verifier. The most straight-forward approach is to model this as the function $\mathsf{zkp}(\mathsf{crs}, x)$. Just as in the other models we have looked at in section 3, we also consider it necessary to include the witness w inside this function even though it is not strictly included in the definition of a zero-knowledge proof. The reason behind this is that the witness has to be present in order for the verification process to work (see below for details on the modeling of the verifier)[20].

Since the algorithm P is probabilistic, we also need to model the use of randomness in this function. We do this by introducing the additional random parameter r. In other words we define a zero-knowledge proof as the 4-arity function $\mathsf{zkp}(\mathsf{crs}, x, w, r)$.

As we have mentioned in 2.2 and have seen in some of the examples provided in 3.1, a statement and/or witness may consist of multiple terms. What does this mean for our zero-knowledge proof function? Since we did not implement zero-knowledge proofs in a Tamarin built-in, it is actually not an issue that the arity may vary between zero-knowledge proofs. For each zero-knowledge proof used in a model one may simply define the function `zkp` according to the needed arity. Another option could be to implement x and w as tuples.

Tamarin even provides the syntactic sugar that tuples can be written as `<t1, ..., tN>`. In other words, the zero-knowledge proof function could easily be declared with fixed arity 4, i.e. `zkp/4`, but then be considered to look like `zkp(crs, <x1, ..., xN>, <w1, ..., wM>)`, where N refers to the number of statement and M the number of witness terms.

We also want to mention a particular way how we handle the witness w. Typically, the witness would be modeled as a persistent fact, i.e. `!Witness(P, w)`. This allows us to then use this fact in a rule which models the adversary compromising an agent and thus revealing all the agent's secrets, including the witness (we will look at this in more detail further down). In some protocols (including our case study which we present in section 5) the witness is created inside the protocol run. In such a case, we would typically add the witness as a persistent fact on the right-hand side of the rule in which it was either created or received. In other cases, the witness might be created by a beacon before the protocol is even initialised. A good example of this is if the witness is a secret key. We use this in our examples in section 4.3. While the witness as a persistent fact does not necessarily reflect the situation in the real-world, it also does not impact neither the adversary's nor any other protocol participants' capabilities, as long as these persistent facts are not used for any other purposes.

Modeling the verifier and the witness relation

Just as with the prover, we can model the probabilistic algorithm V representing the verifier as a verification function. In the formal definition given in section 2.2 the verifier is defined as $V(x, \mathrm{CRS}, P(x, \mathrm{CRS}))$, whereas $V(.) = 1$ iff $\exists w.\ (x, w) \in \mathcal{R}$.

We introduce a term function $\mathsf{verifyZkp}(\mathsf{crs}, \mathsf{zkp}(\mathsf{crs}, x, w, r))$, which mirrors

[20]In sections 6.1 and 6.2 we will propose improvements to this initial model, in which we can in some cases successfully eliminate the parameter w from the zero-knowledge proof function.

the verifier algorithm. We do not add the public statement x as a parameter by itself, as we will introduce an equation to extract the public component. It is therefore not necessary to add it separately. In order to model the verifier's output, i.e. whether he rejects or accepts the proof, we introduce – potentially multiple – equational rules. Each rule will evaluate to the 0-ary term trueZkp[21] if it holds that $(x, w) \in \mathcal{R}$ when that particular rule is applied. As a set, all equational rules must correctly model the behaviour that the function reduces to trueZkp if *and only if* $(x, w) \in \mathcal{R}$.

As we have mentioned in section 2.2, only arguing regarding the membership of (x, w) in $\mathcal{R}$ is not enough to model the verifier's behaviour in the symbolic model. We therefore also require that in each equational rule we specify, the parameter crs used in verifyZkp(.) matches the one used in the function zkp(.).

While for simple examples it is quite straight-forward what this equational rule set is supposed to look like (see for example the equational rule for our running example which we present in section 4.3), it can be very tricky to determine the correct set for more complex zero-knowledge proofs. We will look at this more closely through the examples in this section.

We do need something else however, to ensure that only proofs for which the verifyZkp(.) function reduces to trueZkp are also verified, we employ a technique outlined in the Tamarin manual [15], where it is used to verify signatures. We introduce the following restriction:

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

We can then add the following action to the verifier's protocol rule:

```
Eq(verifyZkp(crs, zkp(crs, x, w, r)), trueZkp)
```

This means that only protocol runs where this equality is indeed fulfilled are accepted. We can thus correctly model the verifier's behaviour.

Modeling the common reference string

As we have seen, we have included the common reference string, i.e. crs as a parameter to the zero-knowledge proof zkp(.) and the verification function verifyZkp(.). What we have to do however, is model the generation and distribution of this parameter. We propose to do this similarly to how keys are typically handled in Tamarin theories (see [15] for examples). This can be compared to a kind of beacon which generates and distributes a fresh common reference string whenever necessary.

We therefore introduce the following rule to model this beacon:

[21]The Tamarin builtins `signing` and `revealing-signing` specify the 0-ary term `true` in order to model their verification processes. We could use this existing constant. However, we consider it better programming practice to separate these two terms clearly by specifying a zero-knowledge specific term with a distinct name in order to avoid any confusion.

```
rule generate_crs:
   [ Fr(~crs) ]
 --[ HonestCRS(~crs) ]->
   [ Out(~crs) ]
```

We consider any common reference string which has been generated through this beacon to have been honestly generated. We model this by adding the action `HonestCRS` in the rule above. This action helps us to ensure that honest parties only use honestly generated common reference strings.

Modeling the adversary's capabilities

Any protocol participant, including the adversary, may retrieve the public component from a zero-knowledge proof. We model this with the term function $\mathsf{pubParams}(\mathsf{zkp}(.))$ and the accompanying simple equational rule:

$$\mathsf{pubParams}(\mathsf{zkp}(\mathsf{crs}, x, w, r)) = x$$

In case the statement x corresponds to a tuple, we advise to declare an equational rule $\mathsf{pubParams}\_i$ for each x_i, such that it reduces to the term at the i-th position in the tuple x. We also want to point out that as we consider the underlying zero-knowledge cryptosystem to be perfectly secure, we do not need to model whether or not the adversary can extract the witness w. This is a reasonable abstraction to make in the symbolic model.

This function does not normally have to be included anywhere in the protocol beyond its declaration, this includes any lemmas. We do however need to model this capability of the adversary (or any other participant) in order to make valid statements.

Let us assume for example the (nonsensical) zero-knowledge proof in Tamarin where the statement x is the tuple `<pk(sk), aenc(m, sk)>` (with the function `pk(sk)` denoting the public key corresponding to the secret key `sk` and `aenc` the asymmetric encryption function provided by the Tamarin built-in `asymmetric-encryption`) and `m` serves as the witness. An adversary could then derive this tuple and by applying built-in Tamarin equational theories could easily deduce the witness `m`. This clearly violates the zero-knowledge property of the proof, but can only be correctly modeled due to the presence of the above equational rule.

While in this example it is obvious that this property would be violated, in a more complex zero-knowledge proof protocol – especially one involving multiple steps – such a deduction will most likely be less obvious. The function $\mathsf{pubParams}$ is therefore crucial to prove the security of the protocol.

An adversary may also compromise an agent and thus get to know the witness. As is typically done for secret keys, we introduce a reveal rule:

```
rule reveal_witness:
   [ !Witness(P, w) ]
```

```
  --[ Reveal(P) ]->
    [ Out(w) ]
```

For some protocols, where the user may want to clearly distinguish between different witness terms, one can use distinct fact names to refer to these different "types" of witnesses (we have done this in our own model of the case study, which we present in section 5). In this case, we recommend to introduce a separate rule for each of these witness facts. The action `Reveal` may stay the same in each of the rules, as we consider any such action to be a reveal, i.e. a compromise, of an agent.

One rule per witness fact is necessary since some witness terms may be constructed during a protocol run, while other parts of the witness are already known at initialisation. This means that there are moments in a protocol execution where only parts of the complete witness are known. However, the adversary may use some initially known parts in an attack *before* the full witness has been constructed. Only by declaring one rule for each witness fact do we ensure that this capability is accurately captured. Naturally, we also recommend to separate these rules from any other additional rules in which an agent's secrets are revealed.

For example, let us consider a zero-knowledge proof where the witness consists of a message and two secret keys, i.e. the tuple `<m, sk1, sk2>`. We can then produce the corresponding facts `!Message(P, m)` and `!SecretKey(P, sk1)` and `!SecretKey(P, sk2)` and introduce the following rules:

```
rule reveal_message:
   [ !Message(P, m) ]
 --[ Reveal(P) ]->
   [ Out(m) ]

rule reveal_secretKey:
   [ !SecretKey(P, sk) ]
 --[ Reveal(P) ]->
   [ Out(sk) ]
```

Note that the rule `reveal_secretKey` may of course be applied multiple times by the adversary to retrieve both secret keys `sk1` and `sk2`.

As discussed in section 2.2, there are attacks which are possible if the common reference string was not honestly generated. We therefore need to consider how to model the adversary's capability to generate a malicious common reference string. As we will see in the next section 4.2, we only prove security properties where we require the common reference string to have been honestly generated. We therefore do not provide a clear guideline on how to model a malicious CRS, as this heavily depends on what kind of attack on the CRS one would want to include. For example, a malicious common reference string could make it possible to break the soundness or the zero-knowledge property. In either case, one would have to include a specific rule enabling the adversary to break the specific property[22].

[22]We also want to mention that it might be necessary to model the common reference string

4.2 Proving security properties

We can imagine many security properties that could be interesting to prove in a protocol employing zero-knowledge proofs. Which ones exactly – as usual – depends on the security goals of the protocol we are modeling. There are however some properties with regards to zero-knowledge proofs which are consistently relevant: the three properties which must be fulfilled by any zero-knowledge proof, namely soundness, completeness and zero-knowledge (see definiton 2.13 and section 2.2).

We therefore need to construct the lemmas representing these properties in such a way that they can be verified by Tamarin (see 2.1.3). In order to do this, we have to lace the protocol rules with appropriate action facts. These action facts can be understood as predicates to be used inside first-order logic lemmas. We introduce the following predicates and their meaning:

1. $\mathsf{KnowsWitness}(P, \underline{w})$: The prover, i.e. agent P, knows the witness $\underline{w}$.
2. $\mathsf{CreatedZkp}(P, \underline{w}, \mathsf{crs})$: The prover, i.e. agent P, has created a zero-knowledge proof for the witness $\underline{w}$ using the common reference string crs.
3. $\mathsf{ReceivedZkp}(V, id, \underline{w}, \mathsf{crs})$: The verifier, i.e. agent V, with protocol run identifier id has received a zero-knowledge proof for the witness $\underline{w}$ using the common reference string crs.
4. $\mathsf{VerifiedZkp}(V, id, \underline{w}, \mathsf{crs})$: The verifier, i.e. agent V, with protocol run identifier id has accepted a zero-knowledge proof as valid for the witness $\underline{w}$ using the common reference string crs.
5. $\mathsf{Finish}(A, id)$: The agent A with protocol run identifer id has completed its execution inside a protocol run.

As stated in 4.1, in case the witness w is a tuple, Tamarin allows for the syntactic sugar to write this as `<w_1, ..., w_n>`. This means that the predicates defined above can be directly adapted iton Tamarin syntax without any necessary changes.

The predicates directly referring to zero-knowledge proofs, namely $\mathsf{CreatedZkp}(.)$, $\mathsf{ReceivedZkp}(.)$ and $\mathsf{VerifiedZkp}(.)$ are dependent not only on the agent and the witness, but also the common reference string. This allows us to make statements regarding whether or not the common reference string with which the proof was created or verified was honestly generated.

We want to point out that we have defined the predicates around the witness itself. We could of course have chosen to do this differently. However, with the witness having to be secret, it is the part of a zero-knowledge proof around which we want to make and prove our claims.

as a persistent fact, i.e. similarly to the handling of a public key (see [15] for an example), instead of only using the `Out` fact as we propose here. This however does not alter in anyway the general functioning of our model.

In some of the predicates we have additionally included the agent's identifier in a particular protocol run, i.e. *id*. We will explain the meaning and necessity of this below as we define our lemmas. It should be noted that we could include the identifier in all of the above predicates, we have however opted to do this only where it was absolutely necessary to construct the lemmas.

There are two more predicates we need in order to construct the lemmas: we have seen these already in section 4.1 when we introduced the rule to generate the common reference string and to model the adversary's capability to compromise an agent. These predicates are HonestCRS(crs) and Reveal(A). These state that crs has been honestly generated and that the agent A has been compromised respectively.

As previously mentioned, these predicates will be placed inside the protocol rules as action facts. The rules in each protocol will of course look very different, since we are in the non-interactive proof scenario. However, there will always be a rule in which the prover sends the zero-knowledge proof, i.e. where the function `Out(zkp(.))` appears on the right-hand side, and one in which a verifier receives the proof, with `In(zkp(.))` on the left-hand side. Also, at one point the witness will either be created or retrieved by the prover P and another point in which the verifier V has all the information required to verify the proof[23].

We therefore can generally put the action `KnowsWitness(P, w)` right after the prover P has created or retrieved the witness. `CreatedZkp(P, w, crs)` can be placed in the rule in which P sends the zero-knowledge proof. Similarly, the action `ReceivedZkp(V, id, w, crs)` is then placed in the verifier's V protocol rule, in which he receives the proof. We are much freer regarding the placement of `VerifiedZkp(V, id, w, crs)`. This can happen at any point after the verifier has received the proof and is in a position to verify it. It also has to appear alongside the action which ensures that the verification function reduces to trueZkp, i.e. `Eq(verifyZkp(.), trueZkp)` (see section 4.1). The last action `Finish(A, id)` can be placed for both the prover and the verifier, but must definitely exist for the latter for reasons we will explain when we define the lemmas. Naturally, this action should be placed when an agent completes its role in the protocol.

For example, following the placement rules above, this could result in the sequence of action facts we see below. Action facts which appear within the same rule instance are put inside brackets.

[23]Depending on the protocol, this does not necessarily have to coincide with the moment at which V receives the proof. An example for this is our case study which we present in section 5.

$$\ldots \rightarrow \texttt{HonestCRS(crs)} \rightarrow \Big[\texttt{KnowsWitness(P, w)} \rightarrow \texttt{CreatedZkp(P, w, crs)}$$
$$\rightarrow \texttt{Finish(P, id1)}\Big] \rightarrow \texttt{ReceivedZkp(V, id2, w, crs)}$$
$$\rightarrow \Big[\texttt{VerifiedZkp(V, id2, w, crs)}, \texttt{Eq(.)}\Big] \rightarrow \texttt{Finish(V, id2)} \rightarrow \ldots$$

Naturally, the user may add any additional action facts to the protocol rules as needed. This generally will not impact the proof of the lemmas we provide here.

We can now define the desired lemmas.

Definition 4.1 ((Knowledge) soundness lemma)**.** The **knowledge soundness lemma** is defined as follows:

$$\begin{aligned}
&\forall\ V\ id\ i_1\ i_2\ w_1 ... w_n \mathsf{crs}.\mathsf{VerifiedZkp}(V, id, \langle w_1, ..., w_n\rangle, \mathsf{crs})@i_1 \wedge \mathsf{HonestCRS}(\mathsf{crs})@i_2\\
&\qquad \implies (\exists\ P\ j.\ \mathsf{KnowsWitness}(P, \langle w_1, ..., w_n\rangle)@j)\\
&\qquad \vee (\exists\ P\ j.\mathsf{Reveal}(P)@j)\\
&\qquad \vee (\exists\ j_1 ... j_n.\ \mathsf{KU}(w_1)@j_1 \wedge\ \ldots\ \wedge \mathsf{KU}(w_n)@j_n)
\end{aligned}$$

The soundness lemma states that if any verifier accepts a zero-knowledge proof which was constructed using an honest common reference string, then there must exist a prover who knows the witness $w_1 ... w_n$ or an agent was compromised (i.e. her secrets, including the witness were revealed) or the adversary has in some other way gained knowledge of $w_1 ... w_n$. The last part models the fact that if the adversary has learnt all the terms which constitute the witness, then he can construct a proof using that witness, therefore the predicate KnowsWitness(.) can no longer hold[24]. As we discussed in section 2.2, there sometimes is made a distinction between knowledge and existential soundness. As we have stated there, we will mainly focus on knowledge soundness in this thesis. The lemma presented here therefore refers to knowledge soundness, as is already suggested by the predicate name KnowsWitness(.). We want to note as well that, as was mentioned in section 4.1, using the model we proposed there, it is in fact not possible to prove strictly existential soundness as we require a fixed w to be present in the proof in order for the equational rule to be able to reduce to trueZkp.

Definition 4.2 (Completeness lemma)**.** The **completeness lemma** is defined

[24]We will revisit this requirement when we look at an alternative formulation of the soundness lemma in section 6.2.

as follows:

$$\begin{aligned}
&\forall\ V\ w_1 ... w_n\ i_1\ i_2\ i_3\ id\ \mathsf{crs}.\\
&\qquad (\exists\ P\ j.\ \mathsf{KnowsWitness}(P, \langle w_1, ..., w_n\rangle)@j)\\
&\qquad \wedge \mathsf{ReceivedZkp}(V, id, \langle w_1, ..., w_n\rangle, \mathsf{crs})@i_1\\
&\qquad \wedge \mathsf{HonestCRS}(\mathsf{crs})@i_2\\
&\qquad \wedge \mathsf{Finish}(V, id)@i_3\\
&\qquad\quad \implies \exists\ k.\ \mathsf{VerifiedZkp}(V, id, \langle w_1, ..., w_n\rangle, \mathsf{crs})@k
\end{aligned}$$

The completeness lemma states that any verifier which has received a valid zero-knowledge proof which was constructed using an honest common reference string must also accept it. It should be noted that this lemma in our model corresponds to a *liveliness* lemma, i.e. it in effect proves whether or not the verifier V is "alive" and able to terminate the protocol. This is due to how we model the verification process: it is impossible for the verifier to not accept a true statement. For this reason it is also necessary to first of all include the predicate $\mathsf{Finish}(V, id)$ in the lemma, but also bind this to the agent's identifier in this particular protocol run. Without this, the lemma can trivially be broken as no protocol participant is required to finish. In other words, just because the verifier has received a zero-knowledge proof, doesn't mean that he also has to indeed accept it, even though it is a valid statement. Since the agent's identifier is re-initiated for each protocol run, we also need it to bind the termination of the protocol to that particular execution.

As we make the assumption that the underlying cryptosystem is perfectly secure, both the soundness and completeness lemmas somewhat trivially hold. However, they should still be included as they can give us an indication whether or not the zero-knowledge proof has been modeled correctly. They therefore can be considered a kind of sanity check of the protocol. We also consider them to be an integral part of the properties we need to guarantee in the symbolic model: with the cryptosystem providing us with a guarantee that these properties hold computationally (under the perfect cryptography assumption), we must also provide a proof that they hold in the symbolic model.

Definition 4.3 (Zero-knowledge lemma)**.**

$$\begin{aligned}
&\neg(\exists\ P\ w_1 ... w_n\ \mathsf{crs}\ i_1\ i_2.\\
&\qquad \mathsf{HonestCRS}(\mathsf{crs})@i_1\\
&\qquad \wedge \mathsf{CreatedZkp}(P, \langle w_1, ..., w_n\rangle, \mathsf{crs})@i_2\\
&\qquad \wedge (\exists\ j.\ \mathsf{K}(w_1)@j \vee\ ...\ \vee \exists\ j.\ \mathsf{K}(w_n)@j)\\
&\qquad \wedge \neg(\exists\ j.\ \mathsf{Reveal}(P)@j))
\end{aligned}$$

This lemma states that the adversary cannot deduce any information regarding the witness without compromising the prover. The actual property we defined in section 2.2 however, does not make a statement regarding the witness, but instead states that the adversary – or indeed any honest participant

– gains no additional knowledge from the proof beyond what he could have computed himself. Unfortunately, this is precisely the kind of property that is hard to represent in the symbolic model as we somehow have to define what this "no additional knowledge" is. In other words, this property cannot be directly translated into a lemma which uses terms and simple protocol predicates. We think however, that the most crucial part is that nothing about the witness is revealed. If in a particular protocol it is the case that there are other pieces of information which may not be deduced, we propose that they are handled similarly to a witness. I.e. they can be kept as persistent facts and can then be included inside the reveal rule. They should then be added as additional "K facts" to the above lemma. In case the witness consists of multiple terms, we of course state that the adversary may not gain knowledge of any of these terms.

As we mentioned before, the three lemmas defined here are crucial properties to prove for any protocol involving zero-knowledge proofs, but they are by far not the most interesting. As we mostly employ these sorts of proofs to provide anonymity or perhaps secrecy for some values, we would want to add some other lemmas. Naturally, these proofs depend on the specific protocol and its security goals and we therefore do not include any further definitions.

As with any protocol model in Tamarin, one should also include a so-called "executability" lemma, which guarantees that the protocol can indeed be executed as intended. We also want to mention that due to how we modeled the verification process if the equational rule set for the verification function is not specified correctly, a well-chosen executability lemma should fail before any of the others: since traces which do not verify a zero-knowledge proof are not considered and the protocol can therefore not be executed. Again, we do not provide a definition of an executability lemma here, but instead refer the user to examples found in [26]. Even though we do not mention it specifically, we have included and verified such an executability lemma in all of the examples we present in section 4.3 and the case study from section 5.

4.3 Example models

In this section we present several examples – of increasing complexity – in which we apply our zero-knowledge proof model presented in section 4.1. For these initial examples we consider only the simplest of protocols where the prover P sends a single zero-knowledge proof to a verifier V, which then either accepts the proof or rejects it. In other words we focus only on the actual zero-knowledge proof and not its embedding inside a larger protocol. While the latter is of course the more interesting scenario as it reflects real-world cases, through these simple examples we can more easily illustrate the capabilities as well as the limitations of our model. We model a more complex protocol for our case study in section 5.

We can write such a simple protocol in the Alice-and-Bob notation as follows:

$$P \rightarrow V : zkp(.)$$

We will only provide code snippets in this thesis. For the full source code we refer the reader to [26][25]. All Tamarin models can be found there, i.e. not only the ones presented in this section, but also in 5 and 6. The models are organised and labeled accordingly.

Simple example

In section 2.2 we have introduced a simple example for a zero-knowledge proof. We now apply our proposed model to this example. We recall that in this example, Alice wants to prove to Bob that she knows the secret key to a ciphertext. In other words, the statement $x = \mathsf{aenc}(m, \mathsf{pk}(sk))$ and the witness $w = sk$. The corresponding witness relation is defined as follows:

$$\mathcal{R} := \{(x, w) \mid \exists m.\ x = \mathsf{aenc}(m, \mathsf{pk}(w))\}$$

For more details on the example we refer the reader to section 2.2. We now present how to model this zero-knowledge proof. Note that many of the parts we described in section 4.1 are generic and do not need to be adapted to this model. We put special focus on the parts which do require adjustments.

Since this example includes asymmetric encryption we must include the builtin `asymmetric-encryption`:

```
builtins: asymmetric-encryption
```

With the statement x and the witness w only consisting of a single term, the handling of these two parameters will be particularly easy. We can therefore declare the following functions:

```
functions: pk/1,
           zkp/4,
           verifyZkp/2,
           trueZkp/0,
           pubParams/1
```

The function `pk` is not central to the zero-knowledge proof model, but is necessary to model the public key used in this example. We need to declare equations only for two of these functions, namely `verifyZkp` and `pubParams`. For the latter we use the exact equation we provided in the description of our model in section 4.1.

To determine the equational rule(s) for `verifyZkp` we need to consider the witness relation we have defined above. Since this relation only states exactly what the statement x and w must correspond to, this easily translates to the following (single) equational rule:

[25]A copy of the source code can also be found in the personal repository, i.e. [27]

```
equations: verifyZkp(crs, zkp(crs, aenc(m, pk(sk)), sk, r)) = trueZkp
```

We then can add the rule `generate_crs` from our general model to generate the common reference string. Also the equality restriction, i.e. `restriction Equality`, which we need for the verification process, can be added without adjustments. We model the public key infrastructure with the following rule (similarly to the examples given in [15]):

```
rule generate_asymKeyPair:
   [ Fr(~sk) ]
 --[ HonestKey($A, ~sk) ]->
   [ !SecretKey($A, ~sk), !PubKey($A, pk(~sk)) ]

rule get_pubKey:
   [ !PubKey(A, pk(skA)) ]
 -->
   [ Out(pk(skA)) ]
```

We next can initialise the prover and the verifier with two simple rules. As is typically done in Tamarin models, we use a fresh value to provide a unique identifier to both the prover and the verifier for a particular protocol run (see also section 4.2). In case of the prover we also retrieve his secret and public key and add this to the agent's knowledge on the right-hand side.

We can now model the actual protocol run with two simple rules: In the prover's rule, he receives the common reference string and creates a message (modeled as a fresh value). Using these terms he can then construct and send the zero-knowledge proof with the function:

```
Out(zkp(crs, aenc(~m, pk(sk)), sk, r))
```

In the verifier's rule, he receives this proof and the common reference string using the `In(.)` function. We then only need to add the following action fact which ensures that only protocol runs in which the proof is also verified are considered.

```
Eq(verifyZkp(crs, zkp(crs, aenc(m, pk(sk)), sk, r)), trueZkp)
```

We also place all other action facts according to the specifications in section 4.2 along with the three lemmas defined in that same section. All lemmas were successfully verified using this model.

Conjunctions and disjunctions

A natural next step is to model a proof whose statement is a Boolean formula over multiple terms. By doing this we also move closer to the existing models by Backes et al. (see [6], [7] and [8] or section 3).

We start by modeling a conjunction. Building from our running example, we can formulate the realistic zero-knowledge proof in which Alice wants to prove knowledge of two secret keys which can decrypt two ciphertexts. We can define the corresponding witness relation as:

$$\begin{aligned}\mathcal{R} := \{((x_1, x_2), (w_1, w_2)) \mid \\ \exists m_1, m_2.\ x_1 = \mathsf{senc}(m_1, w_1) \wedge x_2 = \mathsf{senc}(m_2, w_2)\}\end{aligned}$$

For simplicity reasons we have opted to use symmetric encryption in this example instead and therefore make use of the Tamarin built-in function `senc` from the builtin `symmetric-encryption` (see [15]).

The first thing we observe is that both the statement and the witness consist of multiple terms. This gives us a nice example of how to handle such cases. As we have mentioned in 4.1, we model our examples with a variable arity for the zero-knowledge proof function instead of using the Tamarin syntactic sugar `<x1, ..., x2>` for tuples. In this case the arity for the function `zkp` is defined as $|\underline{x}| + |\underline{w}| + 2$ to account for the parameters `crs` and `r`. This means that `zkp` in this example has arity 6.

We can therefore declare the same functions as in the previous example with the same arities, except for `zkp`, which we declare as `zkp/6`. The only other exception is the function `pk`. This was used to model a public key corresponding to a secret key. Since we are using symmetric encryption only in our model, this function is not necessary.

As described in section 4.1, we declare two equational rules to retrieve the public parameters:

```
pubParams1(zkp(crs, x1, x2, w1, w2, r)) = x1
pubParams2(zkp(crs, x1, x2, w1, w2, r)) = x2
```

In case we would have opted to model this using tuples, we could have written this as:

```
pubParams1(zkp(crs, <x1, x2>, w, r)) = x1
pubParams2(zkp(crs, <x1, x2>, w, r)) = x2
```

Or simply:

```
pubParams(zkp(crs, x, w, r)) = x
```

We also need to add equational rule(s) for the verification function. As this is a conjunction, this translates easily to a single equational rule:

```
verifyZkp(crs, zkp(crs, senc(m1, sk1), senc(m2, sk2), sk1, sk2, r))
     = trueZkp
```

The fact that we chose two separate terms to represent the messages which are encrypted, i.e. `m1` and `m2`, allows us to model the situation where two distinct messages were chosen, but also the case where `m1 = m2`. The rest of the theory can be written almost identically to the one from the previous example.

As a next example we model the disjunction over two terms. We again build from our running example and formulate the proof where Alice wants to prove that she knows the secret key which can decrypt at least one of two ciphertexts. The corresponding witness relation is therefore:

$$\mathcal{R} := \{((x_1, x_2), w) \mid \\ \exists m_1, m_2.\ x_1 = \mathsf{senc}(m_1, w) \lor x_2 = \mathsf{senc}(m_2, w)\}$$

We can construct this model now in a very similar fashion to the conjunction. The notable difference is that with only the statement, but not the witness being a tuple, we have to declare the zero-knowledge proof as a 5-ary function. In order to model the verification process, we declare the following equational rule set:

```
verifyZkp(crs, zkp(crs, senc(m1, sk1), senc(m2, sk2), sk1, r))
     = trueZkp
verifyZkp(crs, zkp(crs, senc(m1, sk1), senc(m2, sk2), sk2, r))
     = trueZkp
```

While in this example the number of equational rules we have to define is still very manageable, this is no longer the case as we consider slightly more complex, but still very simple examples. We will not look at these examples in detail, and instead refer the reader to [26]. But we do want to point out two observations.

For one, the user has to put quite some thought into the choice of the equational rule set and consider all the possible cases under which the verification function reduces to `trueZkp` – or indeed does not. And second, all of our examples were still quite simple and nonetheless, we were already able to see the potential blow-up of the number of equations required to successfully model the verification process.

We were however able to successfully prove all the security properties defined in section 4.2.

Limitations of the model

So far with the generic model we outlined in 4.1, we have been able to model all zero-knowledge proof examples without a problem. We also were able to prove all lemmas we defined in section 4.2. In this last example, we will now look at a witness relation which cannot be modeled:

$$\mathcal{R} := \{((x_1, x_2, x_3), (w_1, w_2, w_3)) | \\ (x_1 = \mathsf{h}(w_1) \lor x_2 = \mathsf{h}(w_2)) \land (\neg(x_1 = \mathsf{h}(w_1)) \lor x_3 = \mathsf{h}(w_3)\}$$

The function $\mathsf{h}(.)$ in this case corresponds to a cryptographic hash function, which we chose over using encryption in order to keep the example as simple as possible. In order to understand why this particular witness relation cannot be modeled, we construct an assignment table, which can be found in table 10. In

	x_1	x_2	x_3	
1	t_1	t_2	t_3	false
2	t_1	t_2	$\mathsf{h}(w_3)$	false
3	t_1	$\mathsf{h}(w_2)$	t_3	true
4	t_1	$\mathsf{h}(w_2)$	$\mathsf{h}(w_3)$	true
5	$\mathsf{h}(w_1)$	t_2	t_3	false
6	$\mathsf{h}(w_1)$	t_2	$\mathsf{h}(w_3)$	true
7	$\mathsf{h}(w_1)$	$\mathsf{h}(w_2)$	t_3	false
8	$\mathsf{h}(w_1)$	$\mathsf{h}(w_2)$	$\mathsf{h}(w_3)$	true

Table 10: Assignment table for the witness relation which breaks the traditional model

each column we denote what the variable x_i is matched against. By the term t_i we denote that the variable x_i can match against any term *except* for $\mathsf{h}(w_i)$.

And here we can already see the issue: through the use of equational rules we cannot model that an otherwise unrestricted term t is *not* supposed to match with another. And this is a problem with regards to this assignment table. If we look at assignments number 3 and 7, we see that they only differ in the assignment to the variable x_1. However, one evaluates to true and the other to false. In particular, if $x_1 = \mathsf{h}(w_1)$ the verifier must reject the proof. But by using an equation to match x_1 to any term t_1 and then the verifier accepts the proof, we have broken the model (see the full corresponding theory in [26]).

While the zero-knowledge proof we presented here is more of a theoretical example, there exists an even simpler one with a potential real-world application. We consider again the first example of a conjunction we presented in this section, where Alice wants to prove that she knows two secret keys which can decrypt two ciphertexts. We can then extend this example to simply include the condition that the two secret keys must be distinct. The corresponding witness relation is:

$$\begin{aligned}\mathcal{R} := \{((x_1, x_2), (w_1, w_2)) \mid \\ \exists m_1, m_2.\ x_1 = \mathsf{senc}(m_1, w_1) \wedge x_2 = \mathsf{senc}(m_2, w_2) \wedge w_1 \neq w_2\}\end{aligned}$$

We clearly cannot model this additional condition using equational rules only, as this includes a comparison between two terms. If we recall the definiton of term rewriting we presented in section 2.1, definition 2.7, we can see that even if $w_1 = w_2$, there exists a substitution such that the verification function will reduce to $\mathsf{trueZkp}$.

This means that our model clearly is not strong enough to successfully model all zero-knowledge proofs. As stated above, it struggles as soon as we have to encode an *inequality* of two terms. In section 6.1 we will present a variation of this model, which does not have this limitation. The possibility of human error is also quite high in our model due to the complexity of choosing the correct equational rule set for the verification function. As we will see, the new model is much less error-prone.

5 Case study: Direct Anonymous Attestation (DAA)

In order to test our model presented in section 4 on a real-world protocol, we have decided to use the same case study as the model from [7] was tested on: the Direct Anonymous Attestation (DAA) scheme. DAA was first introduced by Brickell et al. in [28]. This scheme enables a ***trusted platform module***, or TPM for short, to authenticate itself remotely, while preserving the privacy of the owner of the module.

The DAA protocol is comprised of two subprotocols: the *join protocol* and the *DAA-sign protocol*. In the join protocol, the TPM is issued a ***certificate*** by an entity called the ***issuer***. The DAA-sign protocol then enables the TPM to sign arbitrary messages using the obtained certificate. Such a signature is then verified by the ***verifier***. The first subprotocol is designed to ensure that the TPM cannot be linked to its subsequently created signatures, even by the issuer. The protocol also includes a mechanism called *rogue-tagging*, which prevents corrupted, i.e. "rogue", TPMs from getting issued certificates and authenticating messages.

In order to achieve anonymity, both subprotocols rely on zero-knowledge proofs. This means that there are multiple (different) zero-knowledge proofs within a single protocol, making this an ideal case study.

The protocol we describe in detail below has been taken from [7]. We have chosen to follow the exact same protocol specification as Backes et al. since we want to replicate the results of the case study.

Setup

Each TPM has a unique identifier *id* and a public-private key-pair called an *endorsement key*. It is assumed that the issuer knows the public part of the endorsement key. We will denote the public and secret part of this key as $\mathsf{pk}(\mathsf{ek}(id))$ and $\mathsf{sk}(\mathsf{ek}(id))$ respectively.

A TPM is able to construct so-called *f-values*, which it derives from a secret seed we denote by daaseed_{id}. An f-value f_{cnt} is defined as $\mathsf{H}(\text{daaseed}_{id}, cnt)$, where H is some hash function and cnt is an internal counter. An f-value can be understood as a virtual identity, with which the TPM can execute the join and the DAA-sign protocol.

Additionally, each issuer and verifier holds a publicly known string called the *basename* with bsn_I referring to the basename of the issuer and bsn_V of the verifier respectively.

Join protocol

In the join protocol, a TPM may obtain a certificate from the issuer for one of its f-values. As we do not want the issuer to learn the value f, in order to achieve unlinkability between the TPM and its certificate, we instead blind f

with a random value v. The TPM then constructs a zero-knowledge proof with the blinded value $\mathsf{blind}(f, v)$ as the public statement and f and v as the witness terms.

The zero-knowledge proof also includes a value ζ_I which is a derivative of the issuer's basename bsn_I and $N_I := \mathsf{daa\_exp}(\zeta_I, f)$. The function $\mathsf{daa\_exp}$ is an exponentiation, which was modeled in [7] as a hash function. The use of these values will be discussed further down when we look at the rogue-tagging functionality.

The statement made in the proof is then that the same value f was used to construct the term $\mathsf{blind}(f, v)$ and N_I. The statement and witness are therefore vectors of terms and defined as:

$$\begin{aligned}\underline{x}_{\text{join}} &:= (\mathsf{blind}(f, v), N_I, \zeta_I)\\ \underline{w}_{\text{join}} &:= (f, v)\end{aligned}$$

We define the corresponding witness relation as:

$$\begin{aligned}\mathcal{R}_{\text{join}} := \{\ &((x_1,x_2,x_3),(w_1,w_2)) \mid\\ &\qquad x_1 = \mathsf{blind}(w_1, w_2) \wedge x_2 = \mathsf{daa\_exp}(x_3, w_1)\ \}\end{aligned}$$

If the issuer accepts the proof, he then signs the blinded f-value with its secret issuer's key sk_I (with corresponding public key $\mathsf{pk}(sk_I)$) and returns this value $x := \mathsf{blindsign}(\mathsf{blind}(f, v), sk_I)$ to the TPM. The TPM can then construct the certificate by unblinding x with the secret random value v, i.e. $cert := \mathsf{unblind}(x, v)$. The certificate is therefore equivalent to a valid blind signature on f and can be used by the TPM to sign messages in the next sub-protocol.

The join protocol however, also needs to ensure that only valid TPMs are issued a certificate. The TPM therefore authenticates itself to the issuer through a challenge-response nonce handshake: the TPM identifies itself to the issuer, which then returns a nonce encrypted with the TPM's public endorsement key. The TPM then proves its identity by returning the hash of the nonce together with the blinded f-value $\mathsf{blind}(f, v)$.

The join protocol then looks as follows in Alice-and-Bob notation:

$$\begin{array}{ll}T \to I: & id, \mathsf{zkp}_{\text{join}}(\mathsf{crs}, \mathsf{blind}(f, v), N_I, \zeta_I, f, v)\\ I \to T: & \mathsf{aenc}(n, \mathsf{pk}(\mathsf{ek}(id)))\\ T \to I: & \mathsf{h}(\langle n, \mathsf{blind}(f, v)\rangle)\\ I \to T: & \mathsf{blindsign}(\mathsf{blind}(f, v), sk_I)\end{array}$$

Here we denote the TPM by T and the issuing entity as I. n is the nonce with which the issuer challenges the TPM in the nonce handshake. With the term id we refer to the identifier of the TPM T.

DAA-sign protocol

After successfully executing the join protocol, the TPM is now in possession of a valid certificate *cert* for its f-value signed by the issuer. The TPM now wants to use this certificate to authenticate a message m to a verifier. In other words, he has to convince the verifier that the sender of the message m holds a valid certificate. However, the TPM cannot simply send its certificate since that would reveal its secret value f. Instead, the TPM can again produce a zero-knowledge proof which shows that it knows a valid certificate. In order to link the message m to the certificate, it is included as part of the public statement inside the zero-knowledge proof (otherwise the protocol would be vulnerable to a simple messsage substitution attack). In that way the proof serves as a sort of signature on the message m with the certificate *cert*, wich we call a *DAA-signature.*

Similarly to what was done in the join protocol, additional values are included inside the zero-knowledge proof which serve again in the rogue-tagging functionality which we will discuss below. We include ζ and define $N := \mathsf{daa\_exp}(\zeta, f)$.

One might wonder, why these terms are not analogously named ζ_V and N_V as was done in the join protocol with regards to the issuer I. The reason is that there are two ways in which this protocol may operate: we can either execute an *anonymous* or a *pseudonymous* DAA-sign. In the anonymous DAA-sign ζ will be a fresh value chosen by the TPM. In the pseudonymous DAA-sign ζ can be understood as ζ_V and is derived deterministically from the verifier's basename bsn_V, analogously to ζ_I in the join protocol. In this case N can be understood as N_V and takes the role of a verifier-specific pseudonym of the TPM.

In order to illustrate this we look at the following example: the TPM creates two signatures – i.e. two zero-knowledge proofs – whereas one includes $N_1 = \mathsf{daa\_exp}(\zeta_1, f)$ and the other $N_2 = \mathsf{daa\_exp}(\zeta_2, f)$. In case ζ is a deterministically chosen derivative of bsn_V, then $N_1 = N_2$, which means that the two signatures can be linked. In case ζ is a fresh value however, there is no way the two signatures can be connected to each other. It should be pointed out that in *neither* case is it possible to link the signatures to the execution of the join protocol nor to any signatures for other verifiers, hence it is just a *verifier-specific* pseudonym.

We can therefore define the zero-knowledge proof used in this subprotocol as:

$$\underline{x}_{\text{sign}} := (N, \zeta, \mathsf{pk}(sk_I), m)$$
$$\underline{w}_{\text{sign}} := (f, cert)$$

Whereas *cert* is defined as $\mathsf{unblind}(\mathsf{blindsign}(\mathsf{blind}(f, v), sk_I), v)$. We will continue to use this defintion of *cert* for brevity. We want to stress however, that when we do this we understand the value f used in *cert* to be the same value as the f used alongside it. We again also define the corresponding witness relation as:

$$\mathcal{R}_{\text{sign}} := \{\ ((x_1,x_2,x_3,x_4),(w_1,w_2))\ |\ \exists\, v.\ x_1 = \mathsf{daa\_exp}(x_2,w_1) \wedge \mathsf{blindver}(w_2,\mathsf{blind}(w_1,v),x_3) = \mathsf{true}\ \}$$

For this definition we had to include a function $\mathsf{blindver}(s,t,\mathsf{pk}(sk))$, which evaluates to true if and only if the signature s is a valid signature of the term t using the secret key sk. As the name of the function suggests, this verfication is done "blindly", i.e. the term t is not revealed through the process.

The DAA-sign protocol then consists of only two steps, i.e. the creation of the zero-knowledge proof on the TPM's side and the verification on the verifier's side. In Alice-and-Bob notation this can be written as follows (we again denote the TPM as T and the verifier as V):

$$T \to V: \quad \mathsf{zkp}(\mathsf{crs},N,\zeta,\mathsf{pk}(sk_I),m,f,cert)$$

Rogue-tagging

Since a TPM is a hardware module comprised of a single chip, it is very difficult to extract private information from it. However, it can still be done. In case a TPM is compromised, an attacker is able to sign arbitrary messages and since a certificate cannot be linked to a join protocol run, this cannot even be traced back to the specific TPM. The adversary may even publish f-values and corresponding certificates online, enabling anyone to fake DAA-signatures.

The protocol therefore includes a way in which compromised TPMs may be handled. This is done through rogue-tagging, which uses two separate lists:

1. A list of revoked TPM *id*s which is maintained by the issuer
2. A *rogue list* which contains all f-values which have been made public, so-called *rogue* f-values

Upon receiving an issuance request by a TPM via a zero-knowledge proof, the issuer then checks the TPM's *id* against its list of revoked *id*s and may refuse the request. Note that this only works because the TPM authenticates itself to the issuer. However, already issued certificates remain valid. This is where the rogue list comes into play: let us consider a list of rogue f-values $F := (f_1, ..., f_n)$. Since the values ζ and N are part of the public statement of the zero-knowledge proof, it can easily be checked whether $f \in F$ by checking whether $N = \text{daa\_exp}(\zeta, f_i)$ for some $i \in [1,n]$. This means that a verifier can easily determine whether a certificate has been marked rogue.

The role of ζ_I and N_I is a bit less specific and has not been handeled in the model from [7]. In the original paper by Brickell et al. [28] the authors mention that the issuer may refuse a certificate request by a TPM which has sent too many requests using different N_I in a short amount of time. The authors point out that what exactly "too many" means should be determined according to a risk policy, which they state is beyond the scope of their paper.

As we will see below, we therefore decided, just as [7], not to model the behaviour of the issuer according to the N_I specifically. We however include N_I and ζ_I in the model of the zero-knowledge proof.

The model

For our model we have combined both subprotocols into a single Tamarin theory, which can be found in [26]. As mentioned above, there are two versions of this protocol, an anonymous and a pseudonymous one. We have opted to only model the pseudonymous version. For the anonymous version, we would only have to replace the value of ζ with a fresh value.

Since the protocol uses asymmetric encryption and a hash function, we include the Tamarin built-ins `asymmetric-encryption` and `hashing`. However, there exists no Tamarin built-in which models blind signatures, we therefore add the following functions to our model:

```
blind/2,
blindsign/2,
unblind/2,
blindmsg/1,
blindver/3,
true/0
```

And the single additional equational rule:

```
blindver(unblind(blindsign(blind(f, v), sk), v), blind(f, v), pk(sk))
    = true,
blindmsg(unblind(blindsign(blind(f, v), sk), v)) = f
```

As we have often done in section 4.3, we model the public key as a one-way function `pk` of the secret key. We do want to point out that it is not necessary to model this as a function of the agent, i.e. the TPM's *id*. This is because just as we did in section 4.3, we can model the link between a certain secret key or its corresponding secret key using a rule which generates the public-private key-pair (i.e. simulating a public key infrastructure) and persistent facts. We refer the reader to this previous section for more details.

The DAA protocol also makes use of other operations. In particular, we need to focus on the creation of the f-values, ζ and N.

As previously mentioned, an f-value is defined as $f_{cnt} := \mathsf{H}(\text{daaseed}_{id}, cnt)$, where H is some hash function. We model this as the one-way function (i.e. a function without an equational rule) `daa_h/2`[26]. In order to more closely

[26]We could have also modeled this using the Tamarin built-in hash function `h`. We decided to model this separately simply to highlight that it is a specific function which is described in the original paper [28].

resemble the protocol specifications, we model the seed $\mathrm{daaseed}_{id}$ as a persistent fact `!Seed(T, seed)`, which is generated – similarly to the private-public key-pair – in a separate rule `generate_seed`. The term `seed` is then added to a TPM's state upon initialisation and used to generate theoretically any but practically only one of the TPM's f-values.

The second parameter needed to calculate an f-value is a counter variable in the protocol. We chose to model this as a fresh value. However, this does not fully capture the nature of the counter variable. If the adversary learns the current value of *cnt* and the $\mathrm{daaseed}_{id}$, she could determine all previously created f-values of that TPM and therefore break anonymity. We model this capability by introducing the persistent fact `!Count(T, cnt)` to the right-hand side of the rule in which the f-value was created. Just as for every other secret, we introduce rules enabling the adversary to learn `cnt` from this persistent fact. In that way, all previously used values for *cnt* can be learnt by the adversary in case she compromises a TPM.

As we mentioned before, we model the pseudonymous version of the protocol. This means that both ζ_I and ζ are derivatives of the issuer's and verifier's basename respectively. We model this deterministic derivation in the same way as described in [7]: as a hash on the pairing of the basename and a public constant, i.e. `zetaI = h(<'cnst', bsnI>)` and `zeta = h(<'cnst', bsnV>)`, with `bsnI` and `bsnV` referring to bsn_I and bsn_V respectively.[27]

Similarly to how we introduced the function `daa_h` to model the hash function H, we introduce `daa_exp/2` to model the exponentiation used in the computation of N_I and N. As stated in [28], in the DAA scheme, ζ_I and ζ are chosen in such a way that the computation of the discrete logarithm is infeasible, which means that we may simple model this as a one-way function.

We do want to mention that the protocol specification does not include any exchange of identifiers beyond the TPM's *id* sent alongside his join request. In particular, this means that according to the specification, the TPM does not actually know which I and which V he talks to. So the question that naturally arises is how does the TPM know whose basename to use in the construction of ζ_I and ζ. For simplicity reasons we are excluding this question from our model, especially since this has no impact on the properties we will verify on this protocol. We do want to stress however, that in case one wants to prove linkability properties, this would have to be modeled differently. We refer the reader to [23], where such linkability properties were modeled using Tamarin on the ECC-variant of a DAA protocol.

Next, we need to model the zero-knowledge proofs and their verification

[27]While ζ is derived deterministically in the pseudonymous setting, it is not a fixed value. The authors of [28] mention that how frequently the value of ζ changes will be decided based on a risk policy. They argue that by changing ζ frequently enough, anonymity can also be achieved in this way. However, we think the changing nature of ζ should not be included in the symbolic model and instead recommend to either model the pseudonymous setting with a fixed ζ or the anonymous one where ζ is a fresh value.

process. As we described in detail in section 4, we can model the zero-knowledge proof as a $(|\underline{x}| + |\underline{w}| + 2)$-ary function. As we use two different proofs inside the same protocol, we declare the two corresponding functions separately as: `zkp_join/7` and `zkp_sign/8`.

Both witness relations can be understood as simple conjunctions of predicates and as we have seen in the conjunction example from 4.3, we can model this using a single equational rule for each proof. Since the verification function `verifyZkp` does not have variable arity, we can simply declare one and add the two distinct equational rules:

```
verifyZkp(crs,
   zkp_join(crs, blind(f,v), daa_exp(z,f), z, f, v, r)) = trueZKP
verifyZKP(crs,
   zkp_sign(crs, daa_exp(zeta,f), zeta, pk(sk), m, f,
   unblind(blindsign(blind(f,v), sk), v), r)) = trueZKP
```

We want to point out that the second equational rule is not a direct representation of the witness relation of the DAA-sign protocol. The witness relation contains the predicate $\exists\ v.\ \mathsf{blindver}(w_2, \mathsf{blind}(w_1, v), x_3) = \mathsf{true}$. But $\mathsf{blindver}$ is another function and can therefore not be included inside the equational rule. However, if we want $\mathsf{blindver}$ to reduce to true, this is equivalent to all of the equations below holding:

$$\begin{aligned} w_2 =& \ \mathsf{unblind}(\mathsf{blindsign}(\mathsf{blind}(f, v), sk), v) \\ w_1 =& f \\ x_3 =& \ \mathsf{pk}(sk) \end{aligned}$$

As we can see, this is clearly captured by the equational rule we described above.

What is still missing from the protocol model is the rogue-tagging mechanism. We first of all, introduce two new predicates: $\mathsf{RogueTPM}(A)$ and $\mathsf{RogueVal}(f)$. The first models the TPM A having been added to the issuer's list of rogue TPMs, while the latter represents the f-value f having been added to the rogue list. As this is connected to the compromise of the TPM A, it would make sense to add the action facts `RogueTPM(A)` and `RogueVal(f)` to the "reveal rules". However, the rogue-tagging only takes place, if the agent's compromise was detected. We therefore introduce for each secret a pair of rules representing an undetected and a detected reveal. In the rule for the latter we then obviously include the aforementioned action facts. We have also specifically made a distinction between a publishing of a certificate or an f-value, where the originating TPM is not known. Therefore in such a rule we would only add the action fact `RogueVal(f)` (for the full set of "reveal rules" we refer the reader to the corresponding Tamarin theory found in [26]):

```
rule publish_fValue:
   [ !FValue(T, f) ]
 --[ Reveal(T), RogueVal(f) ]->
   [ Out(f) ]
```

```
rule detectedReveal_fValue:
   [ !FValue(T, f) ]
 --[ Reveal(T), RogueTPM(T), RogueVal(f) ]->
   [ Out(f) ]

rule undetectedReveal_fValue:
   [ !FValue(T, f) ]
 --[ Reveal(T) ]->
   [ Out(f) ]
```

As we have described in section 4, in case a witness is created during the protocol run, we have to add it as a persistent fact on the right-hand side of the protocol rule in which it was created. We therefore have added the corresponding facts `!FValue(T, f)` and `!Certificate(T, unblind(blindsign(blind(f, v), skI), v))`. We want to point out that while in the real-world scenario, the TPM's *id*, T, would obviously not be included as part of the released certificate nor the f-value, we add it inside this persistent fact as it allows us to model that T has been compromised (see above for an example).

What we then need to do is model the issuer's behaviour with regards to the rogue TPMs and the verifier's behaviour with regards to rogue f-values. We do this by adding actions and then corresponding restrictions.

We add the action `AcceptedJoinRequest(T)` to the issuer's protocol rule in which it has successfully completed the nonce handshake with the TPM. The issuer then only proceeds if the request was made by a TPM which has not previously been marked rogue. We can model this with the restriction:

```
restriction NotRogueTPM:
   "All T #i. AcceptedJoinRequest(T)@i
       ==> not (Ex #j. RogueTPM(T)@j & j < i)"
```

To model the verifier's behaviour, we add the action `AcceptedDAASig(f)`, which is naturally added to the singular verifier's rule in which it verifies the zero-knowledge proof. The accompanying restriction is:

```
restriction NotRogueVal:
   "All f #i. AcceptedDAASig(f)@i
       ==> not (Ex #j. RogueVal(f)@j & j < i)"
```

Analogously to the action fact `AcceptedJoinRequest` we also include the action `Joined(T, unblind(blindsign(blind(f,v), skI), v))` in order to model the agreement between the issuer and the TPM that the TPM has successfully completed his join request.

As we have previously mentioned, we do not model the issuer's behaviour regarding vast amounts of requests from the same TPM. This means that no other additions are necessary to model this mechanism.

In [7], the authors have chosen similar predicates, which they referred to as *rogue* and *rogueid*. They do state that these two predicates depend on the values N, ζ and *id*. It is not entirely clear why they chose to do this and why they did not choose to simply define it according the *id* and the f-value respectively.

There is one additional restriction we need to add to the protocol. As it is specified now, the adversary can simply introduce her own f-value and (dishonest) secret key and then simply create a certificate with which she can sign arbitrary messages m. Since the public key corresponding to the secret key with which the certificate was signed is included as a public parameter, a zero-knowledge proof constructed with this (invalid) certificate is clearly valid. While we do not care that the adversary can construct an f-value, this may not be the case for certificates. We therefore added the restriction that the secret key with which the certificate is signed has been honestly generated:

```
restriction IsHonestIssuerKey:
  "All sk #i. HonestIssuerKey(sk)@i
    ==> (Ex I #j. HonestKey(I, sk)@j)"
```

The action fact `HonestIssuerKey(sk)` can then simply be placed in the verifier's rule where he either accepts or rejects the validity of the DAA-signature. We consider this a reasonable restriction, since in real-life, both the TPM and the verifier would have to have a list of issuers' public keys they recognise, otherwise all sorts of attacks would be possible. The above restriction successfully models the behaviour that a verifier only accepts DAA-signatures which include an issuer's public key he considers to be honest.

Security properties

With the model now complete, we can take a look at the lemmas. Ideally, we do not want to adjust the lemmas as we have defined them in 4.2. Since we now suddenly have two distinct zero-knowledge proofs inside a single protocol, this is unfortunately not entirely the case. While the soundness and completeness lemmas did not need adjustments, the lemma for zero-knowledge did. The reason behind this is that we want to make a statement that *no part* of the witness can be learned by the adversary, even if a zero-knowledge proof was created. This means we have to know the structure of the zero-knowledge proof in order to make this statement (for the other lemmas we merely argued regarding the witness tuple `<w1,...,wn>`, which we short-handed as the singular term `w`). In particular, we have to know the number of witness terms in order to make a general statement. By chance both `zkp_join` and `zkp_sign` contain the same number of witness terms. We nonetheless want to introduce a way this can be handled.

In order to do this, we need a way to distinguish the action facts placed for one zero-knowledge proof from the ones placed for the other. However, we do not want to do this for all the action facts, in order to avoid unnecessary overhead. Instead, we recall the action fact `CreatedZkp(P, w, crs)` we introduced in 4.2. As this action fact states that a zero-knowledge proof has been created, it is therefore ideal to also encode the nature of this proof.

We introduce the modified action fact `CreatedZkp(zkpId, P, w, crs)`, whereas the parameter `zkpId` denotes the identifier of the zero-knowledge proof which was created. After the creation of `zkp_join` we therefore introduce the action fact `CreatedZkp('join', P, w, crs)` and analogously for `zkp_sign`. We can then simply state two separate zero-knowledge lemmas, whereas one refers to `CreatedZkp('join', P, w, crs)` and the other to `CreatedZkp('sign', P, w, crs)`. This is a general method which can easily be applied to any other protocol containing multiple zero-knowledge proofs. Whenever one wants to make a general statement holding true for all zero-knowledge proofs, one can simple make this statement for all possible `zkpIds`.

For this case study we also want to prove another property, namely the authenticity of the TPM inside the join protocol. We have chosen this additional property, as the authors from [7] stated that they found an attack on the DAA protocol using their model. We will describe this attack in the next section. First, we will look at the property defined in [7]:

$$\mathsf{JOINED}(id, cnt, cert) \Rightarrow \mathsf{CERTIFIED}(id)$$

This can be easily translated into the following lemma using our previously defined predicates:

$$\begin{aligned}&\forall\ T\ f\ cert\ i.\\ &\qquad \mathsf{Joined}(T, f, cert)@i\\ &\quad\Rightarrow \exists\ j.\ \mathsf{AcceptedJoinRequest}(T)@j\end{aligned}$$

This very simple property just states that if a TPM T has joined under a certificate *cert*, then it must also have been certified by an issuer, i.e. its join request was accepted.

Results

Unfortunately, of the three zero-knowledge proof related lemmas, only the completeness lemma could be successfully verified. Both soundness and zero-knowledge did not terminate. Upon further inspection, we could deduce that this non-termination was linked to the presence of the equational rule for blindmsg, i.e. `blindmsg(unblind(blindsign(blind(f, v), sk), v)) = f`. With the non-terminating lemmas being the ones which contained a clause linked to the adversary gaining knowledge (see the definitions for the lemmas in 4.2), we suspect that this is the reason why the inclusion of this rule, which gives the adversary more deduction capability, leads to non-termination.

Once we remove this equational rule from our model, we are again able to successfully prove all three lemmas. We still consider this to be somewhat of a successful result for the following reasons. First of all, the cause of the non-termination is not directly linked to the zero-knowledge equational theory.

For another reason, if the zero-knowledge property holds, the adversary will in almost no case be able to obtain the term $\mathsf{unblind}(\mathsf{blindsign}(\mathsf{blind}(f, v), sk), v)$, which he needs in order to deduce the f-value. There are two exceptions. One occurs if the adversary compromises the TPM which holds this certificate. This is only because of how the term appears inside the protocol: it is sent as a witness inside $\mathsf{zkp}_{\text{sign}}$ and appears in the persistent "out fact" `!Certificate(.)`, but nowhere else. To model what the adversary can do with `!Certificate(.)`, we already have the following (strong) rules in place:

```
rule publish_certificate:
    [ !Certificate(T, unblind(blindsign(blind(f, v), sk), v)) ]
  --[ Reveal(T), RogueVal(f) ]->
    [ Out(unblind(blindsign(blind(f, v), sk), v)), Out(f) ]

rule dectedReveal_certificate:
    [ !Certificate(T, unblind(blindsign(blind(f, v), sk), v)) ]
  --[ Reveal(T), RogueTPM(T), RogueVal(f) ]->
    [ Out(unblind(blindsign(blind(f, v), sk), v)), Out(f) ]

rule undectedReveal_certificate:
    [ !Certificate(T, unblind(blindsign(blind(f, v), sk), v)) ]
  --[ Reveal(T), UndetectedReveal(T) ]->
    [ Out(unblind(blindsign(blind(f, v), sk), v)), Out(f) ]
```

The second exception occurs if the adversary compromises the TPM and retrieves the random value v used to "blind" the f-value. We have not introduced a rule to model this capability. It makes sense however, that the f-value only remains secret until the originating TPM has been compromised and there already exists a rule which models the reveal of v.

With these rules, we have already given the adversary the capability of obtaining f from the value $unblind(blindsign(blind(f, v), sk), v)$, namely by adding the fact `Out(f)` to the right-hand side of the rule.

In other words, by having proven that the zero-knowledge property holds, we can confidently say that all three lemmas hold. We therefore conclude that we were successfully able to apply our model to a real-world protocol.

The question which is left open is whether or not we can also use our model to prove other properties. In particular, we want to see if we can replicate the attack presented in [7]. This attack is on the authenticity of the TPM inside the join protocol. As we recall, the TPM authenticates itself to the issuer with a challenge-response nonce handshake. This is vulnerable to a man-in-the-middle attack. Here the adversary corrupts the secret endorsement key of a TPM A. Another (uncorrupted) TPM B then sends a join request to the issuer. The adversary interrupts this request, replaces the correct TPM's identifier B with the one of the corrupted TPM A. The issuer then conducts the challenge-response handshake using the corrupted endorsement key of A, i.e. the adversary sits in the middle. We can write this attack in the Alice-and-Bob notation (we denote the adversary posing as the issuer by $I(A)$):

$$\begin{aligned}
B \rightarrow I(A): &\quad B, \mathsf{zkp\_join}(...\mathsf{blind}(f,v)...)\\
A \rightarrow I: &\quad A, \mathsf{zkp\_join}(...\mathsf{blind}(f,v)...)\\
I \rightarrow A: &\quad \mathsf{aenc}(n, \mathsf{pk}(\mathsf{ek}(A)))\\
I(A) \rightarrow B: &\quad \mathsf{aenc}(n, \mathsf{pk}(\mathsf{ek}(B)))\\
B \rightarrow I: &\quad \mathsf{h}(\mathsf{blind}(f,v), n)\\
I \rightarrow B: &\quad \mathsf{blindsign}(\mathsf{blind}(f,v), sk_I)
\end{aligned}$$

In the end, it is only B which holds the certificate: due to the zero-knowledge property, the adversary cannot gain any knowledge regarding the secret components of the `zkp_join` function (also the adversary having already corrupted the secret endorsement key of another TPM with which it could obtain a certificate has nothing to gain from this). However, B receives the certificate without having been authenticated by the issuer. B could be tagged as rogue or the issuer may want to refuse to issue a certificate to B for another reason. This therefore violates the security goals of the protocol.

Using the corresponding lemma we defined above for our protocol, we were able to replicate this attack. We then decided to also implement the simple protocol fix suggested by [7]: instead of sending the TPM's *id* alongside the join request, the *id* gets added to the public component of the zero-knowledge proof. We again provide the full Tamarin theory in [26].

However, some attacks are still possible. First of all, if an issuer is compromised, the adversary can trivially sign arbitrary join requests without having authenticated the TPM. We consider this to not be a consequential attack, since this property can obviously only be fulfilled until the issuer has been compromised. Indeed, it is not clear why the lemma in [7] was not adjusted accordingly, as it clearly cannot hold.

But even if we add that authenticity can only hold until the adversary has compromised an issuer, an attack similar to the one described by the authors is unfortunately still possible. In this case, the adversary additionally obtains the f-value of B by compromising him. She can therefore construct her own proof and can successfully inject the compromised agent's A identifier into the public component of the proof[28].

This attack is possible for two reasons: for one we differentiate between an undetected and a detected reveal and have therefore made the adversary stronger. Since we were unable to obtain a copy of the source code of the model from [7], we cannot compare this to the capabilities of their adversary. However, it does seem reasonable that authenticity only holds unless an f-value was corrupted and not tagged rogue. The other reason for which the attack is possible is that the issuer does not check whether or not an f-value with which a TPM issues a join request has been tagged rogue. The issuer therefore reissues a certificate for a rogue f-value. This certificate is quite useless however since

[28]For details of the attack we refer the reader to the attack diagram found in [26]

it already has been made invalid through the rogue tag. We therefore have decided to adjust the lemma to the following:

$$
\begin{aligned}
&\forall\ T\ f\ cert\ i.\\
&\quad \mathsf{Joined}(T, f, cert)@i \land \neg(\exists\ I\ k.\ \mathsf{RevealIssuer}(I)@k)\\
&\quad \land \neg(\exists\ k.\ \mathsf{UndectedRogueVal}(f)@k)\\
&\quad \land \neg(\exists\ k.\ \mathsf{RogueVal}(f)@k)\\
&\Rightarrow \exists\ j.\ \mathsf{AcceptedJoinRequest}(T)@j
\end{aligned}
$$

We were able to successfully prove the lemma with the above adjustment on the fixed protocol according to the suggestion from [7] (and without the equational rule for blindmsg[29]). This means that our model was also successful in proving additional protocol properties.

[29]Due to the RevealIssuer(.) action fact, the verification of the lemma would otherwise again not terminate.

6 Alternative models

In section 4 we have introduced a general model for zero-knowledge proofs, which we have applied to a series of simple examples in 4.3 and our case study protocol in section 5.

While our model succeeds in modeling real-world cases, we also identified some down sides to using this straight-forward, traditional approach to modeling cryptographic primitives in Tamarin. In this section we will therefore explore two alternative approaches to this "traditional" model. We first introduce what we call a "restriction-only" model and then give alternative definitions of the soundness, completeness and zero-knowledge lemmas we presented in 4.2.

6.1 Restriction-only model

We mainly want to address two major issues we found with our "traditional model". First of all, we came across two examples in 4.3, which could not be modeled with our approach. The natural question that arises is whether this is a limitation inherent in the symbolic model or could we eliminate this by altering our model.

The second disadvantage we found is that even with regards to the proofs that can be modeled, we have to encode a potentially complex verification process inside sometimes multiple term equations. This is error-prone and, depending on the proof's complexity, introduces many maintainability issues. As the goal of proving properties in the symbolic model is precisely to limit human error and create more reliable proofs, this misses the mark.

The reason for this added complexity is that we are actually encoding the definition of the witness relation into these multiple equations. A witness relation is nothing else but a Boolean formula over predicates which have to hold for a pair $(\underline{x}, \underline{w})$. It is therefore not surprising why we did not – and indeed cannot – always represent a Boolean formula in a set of equational rules.

In this section we propose a new way to encode the witness relation and therefore not only immensely reduce the complexity of the specification of the verification process, but are also able to successfully model our counter-examples.

In the general model introduced in section 4 we have already come across the *restriction* functionality provided by Tamarin. As stated in [15], it "restrict[s] the set of traces to be considered in the protocol analysis" and is therefore particularly useful when it comes to modeling the internal behaviour of a protocol participant. The manual specifically recommends to use restrictions during the verification process of signatures when using the built-in theory for signing. Examples are given where an "equality restriction" is defined, such that only protocol traces are considered in which the signature verification function `verify` can be reduced to `true`. We have used this exact mechanism inside our own model. What is interesting about restrictions is that they are written in first-order logic, in other words, we are able to write any Boolean formula inside a restriction. The natural question that therefore arises is whether we can encode the *entire* verification process including the witness relation in a restriction.

When we look at the verification process itself, this is nothing else than the verifier accepting the proof if $(\underline{x}, \underline{w}) \in \mathcal{R}$ and otherwise rejecting it. If we want to translate this into a restriction, this is the same as saying that we restrict all protocol traces in which $(\underline{x}, \underline{w}) \notin \mathcal{R}$. We therefore must only find a way to encode this membership in $\mathcal{R}$ as a Boolean formula. We recall from section 2.2 that formally, the witness relation is defined as:

$$\mathcal{R} := \{ \ (\underline{x}, \underline{w}) \mid \text{Predicates over } \underline{x} \text{ and } \underline{w} \ \}$$

We can then translate the verification process to the following formula:

$$\forall \ \mathsf{crs} \ \underline{x} \ \underline{w} \ i. \ \mathsf{ValidZkp}(\mathsf{crs}, \mathsf{zkp}(\mathsf{crs}, \underline{x}, \underline{w}))@i \\ \implies \text{Predicates over } \underline{x} \text{ and } \underline{w}$$

This formula is of course directly translatable to a syntax suitable for a Tamarin restriction and replaces the previous equality restriction. We can then replace the action fact `Eq(zkp(.), trueZkp)`, with `ValidZkp(crs, zkp(.))`. As we noted in section 4.1, the common reference string `crs` used in the creation of the proof has to be the same as the one used inside the verification. We want to stress that this condition is also fulfilled when using a restriction as specified above.

Our general model described in section 4 can then be further simplified by removing the function terms `verifyZkp` and `trueZkp`, as well as any corresponding equational rules. Instead, we directly translate the witness relation as described above and add the corresponding action and restriction. No further adjustments are necessary.

Due to the sole use of restrictions for the verification process, we call this a *restriction-only model*. With this new approach we have found a way to directly and cleanly encode the witness relation and in many ways more successfully model the behaviour of the verifier algorithm.

We have tested this adapted model on all of our previous examples, including the case study protocol, and were able to successfully model all of them using this restriction-only approach. Below we show the encoding of the witness relation as restrictions for some of the examples from section 4.3. For the full Tamarin theory files, we refer the reader to [26].

Examples

As a first example, we want to again refer to the running example introduced in section 2.2. We recall the witness relation which is of the most simple form:

$$\mathcal{R} := \{(x, w) \mid \exists m. \ x = \mathsf{aenc}(m, \mathsf{pk}(w))\}$$

According to our specifications above, this translates to:

```
restriction IsValidZKP:
   "All crs x w r #i. ValidZKP(crs, zkp(crs, x, w, r))@i
       ==> Ex m. x = aenc(m, pk(w))"
```

As we can see, this is a direct translation of the predicates in the witness relation. If we look at the equational rule we presented in 4.3 for the same zero-knowledge proof, it would not be as straight-forward to infer the witness relation from the equation – even much less so for other examples where we required multiple equations. Whereas using our restriction-based model, the translation backwards is just as straight-forward.

As a next example, we want to consider the zero-knowledge proofs for which we failed to produce a valid model. We recall the following witness relation:

$$\mathcal{R} := \{((x_1, x_2), (w_1, w_2)) \mid \\ \exists m_1, m_2.\ x_1 = \mathsf{senc}(m_1, w_1) \wedge x_2 = \mathsf{senc}(m_2, w_2) \wedge w_1 \neq w_2\}$$

While we could not find a correct representation through an equational rule set, we can now model the verification process as the following restriction:

```
restriction IsValidZKP:
   "All crs x1 x2 w1 w2 r #i.
    ValidZKP(crs, zkp(crs, x1, x2, w1, w2, r))@i
       ==> (Ex m. x1 = senc(m,w1))
         & (Ex m. x2 = senc(m,w2))
         & not w1 = w2"
```

With the traditional model, we could run a protocol execution with a false proof – i.e. where the two secret keys were not distinct – in which the verifier has accepted the proof instead of rejecting it. In our restriction-only model, the "executability lemma" fails, which means that the false proof was rejected as it should have been. When running the restriction-only model with a valid proof, as expected, we can prove all the lemmas defined in section 4.2 as well as the "executability lemma". The same holds for the other example we gave which could not be modeled with our "traditional" model.

As a last example, we want to look at the zero-knowledge proofs from our case study from section 5. In this example we introduced two distinct zero-knowledge proofs inside the same protocol. This of course means that in our restriction-only model we need to define two restrictions to represent the two distinct witness relations. We can of course do this by adding two separate predicates $\mathsf{ValidZkp\_join}(.)$ and $\mathsf{ValidZkp\_sign}(.)$ for the corresponding functions `zkp_join` and `zkp_sign`.

The restriction for the witness relation $\mathcal{R}_{\text{join}}$ is quite straight-forward and we will not further expand on this. The case is slightly more interesting for the witness relation for the DAA-sign zero-knowledge proof, $\mathcal{R}_{\text{sign}}$. Just as with the traditional model we are again not able to directly translate the predicate $\exists\, v.\ \mathsf{blindver}(w_2, \mathsf{blind}(w_1, v), x_3) = \mathsf{true}$. The reason here is probably less obvious. Following the approach we have used so far, we would want to write the restriction like this:

```
restriction IsValidZkp_sign:
   "All crs x1 x2 x3 x4 w1 w2 #i.
    ValidZkp_sign(crs, zkp_sign(crs, x1, x2, x3, x4, w1, w2))@i
      ==> ( x1 = daa_exp(x2, w1)
          & (Ex v sk. blindver(w2, blind(w1, v), x3) = true) )"
```

The issue however is that Tamarin does not allow for reducible function symbols to be used inside restrictions. This therefore leads to a wellformedness error which states that the restriction uses terms of the wrong form. We instead encode the equivalent witness relation we described in section 5. This leads to the restriction:

```
restriction IsValidZkp_sign:
   "All crs x1 x2 x3 x4 w1 w2 #i.
    ValidZkp_sign(crs, zkp_sign(crs, x1, x2, x3, x4, w1, w2))@i
      ==> ( x1 = daa_exp(x2, w1)
          & (Ex v sk. x3 = pk(sk) &
                      w2 = unblind(blindsign(blind(w1, v), sk), v)
            )
          )"
```

We recognise that the only limitation of our model might be that reducible function terms may not be included. However, in all the examples that we have encountered where this was an issue initially, we could find an equivalent encoding such as the one for $\mathcal{R}_{\text{sign}}$.

In conclusion, we were not able to find a counter-example of a zero-knowledge proof which could not be modeled using the restriction-only approach.

6.2 Alternative lemma definition

In addition to the restriction-only model alternative we have discussed in the previous section, we can introduce another alternative to our model, this time with regards to the lemmas.

The reason we propose this alternative is perhaps more subtle than the one for introducing the restriction-only model. In order to construct the zero-knowledge proof lemmas in our model, we rely heavily on action facts which are bound to agents, i.e. the prover and the verifier (see section 4.2 for details). If we now compare this to the mathematical definition of zero-knowledge proofs we have given in section 2.2, we can see that they are not making the exact same statement.

This issue is especially apparent in the definition of the soundness lemma, see definition 4.1. To a reader who is familiar with writing lemmas in Tamarin, the last part of the lemma, i.e. the statement that soundness can only be guaranteed until the adversary has learned the witness, will probably be very familiar. It is common to state secrecy lemmas in a similar way, i.e. one can only guarantee secrecy until an agent has been compromised. We also saw something similar regarding the authenticity lemma we presented for our case study in section 5. However, with regards to zero-knowledge proofs, this does not accurately depict how soundness is defined in the mathematical sense. The last two terms in the above lemma can be understood to represent the adversary's ability to construct proofs herself. We then say that if a statement was verified it was valid unless the statement was made by the adversary. But perhaps somewhat counter-intuitively, we do not care who makes the statement, only that it is valid. So if the adversary learns a valid witness, she may construct a proof for a valid statement and that should just as much be accepted as if it were made by an honest protocol participant.

We therefore propose in this section an alternative version to the definitions provided in section 4.2, namely one that is closer to the mathematical definition.

We will illustrate the general idea behind this alternative definition with the soundness property. If we look at the mathematical definition of soundness from 2.13, at a high-level, this property states that if a statement x is false, the verifier must reject it. We can turn this around to saying that if the verifier has accepted a proof, then its statement must have been valid.

In many ways, this is already reflected in our definition of the soundness lemma (see definition 4.1). The only part we want to get rid of is the statement that soundness only holds for proofs generated by honest agents. The issue is however that we define a valid statement in our lemma as an honest prover existing who knows the witness. So the issue is in fact the way in which we argue about the validity of the statement. We therefore have to find an alternative to how we encode this.

Mathematically a valid statement is defined as there existing a w, such that $(x, w) \in \mathcal{R}$. An obvious approach is therefore to directly encode this into the lemma, i.e. we want to say that if a proof has been verified then there exists a w such that the predicates over $\underline{x}$ and $\underline{w}$ from the witness relation hold. As it turns out, this approach indeed works and fixes our issue.

Before we define the alternative versions of the lemmas, we have to introduce new predicates, which replace the ones we have defined in 4.2:

1. $\mathsf{CreatedZkpForWitness}(P, \underline{w}, \mathsf{crs})$: The prover, i.e. agent P, has created a zero-knowledge proof for the witness $\underline{w}$ using the common reference string crs. This predicate replaces the previous $\mathsf{CreatedZkp}(P, \underline{w}, \mathsf{crs})$.

2. $\mathsf{ReceivedZkp}(V, id, \mathsf{zkp}(.))$: The verifier, i.e. agent V, with protocol run identifier id has received the zero-knowledge proof $\mathsf{zkp}(.)$.

3. $\mathsf{VerifiedZkp}(V, id, \mathsf{zkp}(.))$: The verifier, i.e. agent V, with protocol run identifier id has accepted the zero-knowledge proof $\mathsf{zkp}(.)$ as valid.

4. $\mathsf{Finish}(A, id)$: (unchanged)

We have also made use of another predicate $\mathsf{VerifiedZkpForWitness}(P, \underline{w}, \mathsf{crs})$. However, we only use this inside our executability lemma and we therefore do not define this further.

We can now define the following lemmas:

Definition 6.1 (Alternative (knowledge) soundness lemma)**.** The **alternative knowledge soundness lemma** for a zero-knowledge proof with the corresponding witness relation $\mathcal{R} = \{(\underline{x}, \underline{w}) \mid \text{Predicates over } \underline{x} \text{ and } \underline{w}\}$ is defined as follows:

$$\begin{aligned}
&\forall\ V\ id\ i_1\ i_2\ x_1 ... x_n\ w_1 ... w_m\ r\ \mathsf{crs}.\\
&\qquad \mathsf{VerifiedZkp}(V, id, \mathsf{zkp}(\mathsf{crs}, \langle x_1, ..., x_n\rangle, \langle w_1, ..., w_m\rangle, r))@i_1\\
&\quad \wedge \mathsf{HonestCRS}(\mathsf{crs})@i_2\\
&\quad \Longrightarrow \text{Predicates over } \underline{x} \text{ and } \underline{w}
\end{aligned}$$

Definition 6.2 (Alternative completeness lemma)**.** The **alternative completeness lemma** for a zero-knowledge proof with the corresponding witness relation $\mathcal{R} = \{(\underline{x}, \underline{w}) \mid \text{Predicates over } \underline{x} \text{ and } \underline{w}\}$ is defined as follows:

$$\begin{aligned}
&\forall\ V\ id\ i_1\ i_2\ x_1 ... x_n\ w_1 ... w_m\ r\ \mathsf{crs}.\\
&\qquad (\text{Predicates over } \underline{x} \text{ and } \underline{w})\\
&\quad \wedge \mathsf{ReceivedZkp}(V, id, \mathsf{zkp}(\mathsf{crs}, \langle x_1, ..., x_n\rangle, \langle w_1, ..., w_m\rangle, r))@i_1\\
&\quad \wedge \mathsf{HonestCRS}(\mathsf{crs})@i_2\\
&\quad \wedge \mathsf{Finish}(V, id)@i_3\\
&\qquad \Longrightarrow \exists\ k.\ \mathsf{VerifiedZkp}(V, id, \mathsf{zkp}(\mathsf{crs}, \langle x_1, ..., x_n\rangle, \langle w_1, ..., w_m\rangle, r))@k
\end{aligned}$$

Definition 6.3 (Alternative zero-knowledge lemma)**.** The **alternative zero-knowledge lemma** is defined as follows:

$$\begin{aligned}
&\neg(\exists\ P\ w_1 ... w_m\ \mathsf{crs}\ i_1\ i_2.\\
&\qquad \mathsf{HonestCRS}(\mathsf{crs})@i_1\\
&\quad \wedge \mathsf{CreatedZkpForWitness}(P, \langle w_1, ..., w_n\rangle, \mathsf{crs})@i_2\\
&\quad \wedge (\exists\ j.\ \mathsf{K}(w_1)@j \vee\ ...\ \vee \exists\ j.\ \mathsf{K}(w_n)@j)\\
&\quad \wedge \neg(\exists\ j. \mathsf{Reveal}(P)@j))
\end{aligned}$$

The first thing we want to note is that the soundness and the completeness lemma now have to be defined specific to a zero-knowledge proof, as they contain the definition of their corresponding witness relation. We can also note that the zero-knowledge lemma did actually not change. We merely replaced the old predicate $\mathsf{CreatedZkp}(.)$ with $\mathsf{CreatedZkpForWitness}(.)$. This is due to the fact that it makes reference to the validity of the statement.

As we can see, we have successfully managed to eliminate the requirement that the statement must have been made by an honest prover. What is interesting is that this way of formulating the lemmas actually enables us to finally prove the existential soundness property on its own. For some zero-knowledge proofs we can now remove the witness as a parameter of the zkp function, such that $\mathsf{zkp}(\mathsf{crs}, \underline{x}, r)$. We therefore define the following existential soundness lemma:

Definition 6.4 ((Existential) soundness lemma)**.** The **existential soundness lemma** for a zero-knowledge proof with the corresponding witness relation $\mathcal{R} = \{(\underline{x}, \underline{w}) \mid \text{Predicates over } \underline{x} \text{ and } \underline{w}\}$ is defined as follows:

$$\begin{aligned} &\forall\ V\ id\ i_1\ i_2\ x_1...x_n\ r\ \mathsf{crs}. \\ &\qquad \mathsf{VerifiedZkp}(V, id, \mathsf{zkp}(\mathsf{crs}, \langle x_1, ..., x_n\rangle, r))@i_1 \\ &\quad \wedge \mathsf{HonestCRS}(\mathsf{crs})@i_2 \\ &\Longrightarrow \exists\ w_1...w_m.\ \text{Predicates over } \underline{x} \text{ and } \underline{w} \end{aligned}$$

Since we have removed the witness from the zero-knowledge proof function, we also have to adjust the completeness lemma analogously. We have therefore managed to formulate lemmas which are as close as we can possibly get to the mathematical definitions of zero-knowledge proofs.

Examples

We will not look at any examples in detail and instead refer the reader to the Tamarin theories provided in [26], in the directory corresponding to this section (i.e. section 6.2)[30]. We have implemented this model based on the restriction-only model we introduced in section 6.1, we want to point out however that these alternative lemmas can also be applied to the traditional model from section 4.

There is however one particularity we want to mention. We also applied this modification of our traditional model to the case study protocol we introduced in 5. We recall that this protocol includes two distinct zero-knowledge proofs. As we have stated before, we are now – theoretically – able to prove existential soundness instead of knowledge soundness. However, this is not practically applicable to all zero-knowledge proofs. As we can see the $\mathsf{zkp}_{\text{sign}}$ proof cannot be modeled that way. This is due to the fact that there is an inter-dependency between the various terms of the witness, i.e. f corresponds to the first term of the witness and must also be used to construct the second part of the witness, namely the certificate (we refer the reader to section 5 for details). We can however still use the definition for the alternative knowledge soundness lemma we have given above.

With regards to our case study, there was an additional advantage to proving the alternative lemma definitions. We recall that there was an issue regarding non-termination of the proof of the lemmas containing the $\mathsf{Reveal}(.)$ predicate. With this no longer being the case in the alternative soundness lemma, the verification terminated even when including the equational rule for the $\mathsf{blindmsg}(.)$ function.

[30]Some additional alternative lemma formulations can also be found in the Tamarin theory files in the directory corresponding to the previous section 6.1 in [26]. We refer the reader to the accompanying read-me file.

	initial	**restriction-only**	**alternative-lemma**
DAA	6.831	8.8735	1.783
simple	0.161	0.167	0.163
conjunction	0.181	0.187	0.177

Table 11: Runtime in [s] of proofs in Tamarin of various models.

6.3 Discussion of alternative models

In this section, we will evaluate possible advantages or disadvantages the modifications we presented in 6.1 and 6.2 might have over the initial model introduced in section 4.

We have analysed the run-time of several of our models on a 2.3 GHz Quad-core Intel i7 and with Tamarin version 1.6.0. We present the results in table 11. For obvious reasons, we have only run the proofs on the case study variant without the equational rule for blindmsg, as otherwise we would have a non-terminating result. As we can see, for our simple examples, the differences in run-time are only marginal and cannot be considered statistically relevant. With regards to the run-time of our case study, the results are however quite clear. We will now discuss this in further detail.

First of all, we have to mention that the number of lemmas which were proven in the initial, restriction-only and alternative-lemma models are not the same. This is due to the fact that in the alternative-lemma model we had to introduce a separate lemma for each of the zero-knowledge proofs, not only for zero-knowledge but also soundness and completeness. But even though in that model we had to verify more lemmas, the runtime was significantly lower than for the initial model. Whereas the restriction-only model required slightly more runtime. This seems to consistently be the case also for our simple examples, however as we mentioned before, we do not consider these variations to be statistically relevant.

As previously discussed, there are some limitations to our initial model. Nonetheless, there are advantages to this model with regards to runtime. For certain complex protocols it could be beneficial to decrease verification time. This is especially the case when paired with the alternate-lemma model. While our examples all use the restriction-model for the verification process, the alternate-lemma model can also be used together with the initial model. This could increase runtime dramatically.

It might therefore seem strange to even include the original definition of the lemmas. However, there could be proofs where it might not be possible to postulate the lemmas in the required way, i.e. we might not find a way to satisfactorily encode the predicates of the witness relation. As we have mentioned in section 6.1, a possible limitation to the restriction-only model might be the fact that certain function symbols may not appear inside a restriction. This might lead to similar problems in lemmas. Another advantage of the initial definition of the lemmas is that they are more generalisable as they are independent of the specific zero-knowledge proof.

7 Comparing Tamarin to existing models

With the previously existing models we presented in section 3 showing so many vast differences, we want to analyse in this section where our own model from section 4 along with its modifications from section 6 falls.

At first glance, our initial model from section 4 seems to be closest to the model from [8]. We can compare the two zero-knowledge proof functions, $\mathsf{zkp}(\mathsf{crs}, x, w, r)$ (our own) to $\mathsf{ZK}(\mathsf{crs}(N_1), x, w, N_2)$ (from [8]). This comparison becomes even stronger when we look at how we defined the verification process in figure 1 to the one listed in table 8: in both cases do we define equational rules reducing to (some notion of) true under the condition that $(x, w) \in \mathcal{R}$. As we have previously mentioned though, we do not know how the authors of [8] encoded the witness relation. It could be conceivable however, that they also reverted back to defining a set of equational rules. If this is the case, our initial model can be viewed as the Tamarin equivalent to the model presented in [8]. This would also mean that their model must show the same limitations as ours and vice versa. As we were unable to obtain a copy of the source code despite multiple attempts, we can unfortunately not make any further conclusions.

As we have previously discussed there are some obvious limitations to our initial model. These apparently stem from the fact that we use an equational rule set to encode the witness relation instead of a Boolean formula, since we were able to successfully counter these limitations in the restriction-only model. Because of this, it might seem that the model from [7] is stronger than the initial model as it uses the formula F. This however is not always the case. We recall that the authors described a compiler which could convert the zero-knowledge proof specifications into an acceptable ProVerif input. However, they have mentioned that ProVerif would on occasion no longer terminate on the output of their compiler. This is apparently mainly the case due to the presence of the Boolean operators $\wedge$ and $\vee$. For these cases, they described a mechanism with which – under certain conditions – the output could be modified such that ProVerif could again terminate. Unfortunately we could not inspect this thoroughly, since we – as previously mentioned – could not obtain a copy of the source code. However, as far as we could deduce from examples from the paper, this mechanism includes the expansion of an equational rule for the verification process to a *set*. This appears to be quite similar to our encoding of the witness relation in the initial model. It is also not clear if this adjustment has to be done manually or not. If that is the case, then it appears that our initial model is quite similar to the model from [7] and that the latter – apart from the convenience of an initial compiler – possibly has the same short-comings as our model.

The most interesting comparison however is between our restriction-only model to the one presented in [6]. We recall that the latter model included the notion of a "true zero-knowledge proof", which evaluates to *true* or *false* in a logical sense instead of using equational rules. This is very similar to how the verification is done in the restriction-only model, as restrictions are written in first-order logic. Therefore, a statement is also considered valid according to whether or not a Boolean formula evaluates to *true*. The similarity between the models is even stronger if we consider that "true zero-knowledge proofs" need to evaluate to *true* in a syntactical sense. Since we are not allowed to use function symbols for which there exists an equational rule inside restrictions, it appears that the equality of ZKTerms is similar to the terms that may appear in a restriction.

The model from [6] however is much more restrictive regarding the terms allowed inside these "true zero-knowledge proofs". We leave it as an open question to evaluate whether or not the model from [6] could be extended to allow for additional (as well as arbitrary user-defined) terms and whether our restriction-only model then constitutes a valid implementation of that model.

Additionally, there is a strong indication that our restriction-only model in combination with the alternative formulation of the lemmas is no longer as restrictive with regards to the conditions on zero-knowledge proofs presented in [6] and then weakened in [8]. Especially with regards to the *extractability* property which states that in the abstract model one needs to include the witness to construct the proof. As we have shown in section 6.2, this is not always the case.

We also want to discuss the issue we encountered regarding non-termination in our case study. With this being the same case study as in [7] and there having also been some non-termination issues mentioned in that paper, we want to compare this in more detail. From the paper [7] it appears that non-termination was not issue when it came to the model of the case study, however this was not mentioned specifically. However, there is no indication that these appearances of non-termination are connected. Unfortunately, it is not always clear why Tamarin does not terminate. We leave it as an open question to compare the model of the DAA protocol presented here to the ECC-variant modeled in [23] and [22].

In conclusion, our results suggest that our model is stronger and has less limitations than the other models. We can achieve this specifically by modeling the zero-knowledge proofs in Tamarin in which we use its special functionality of restrictions. However, we can make no formal claim regarding the strength of our model at this point. In general, more formalisation is necessary to understand the exact limits of our proposed model and in particular compare it to the strict limitations which were outlined in particular in [6] and [8].

8 Conclusion and future work

We were successfully able to formulate a generic model for zero-knowledge proofs in Tamarin. We have both applied this model to various simple examples as well as a real-world protocol in our case study. Together with the possible modifications of the model we presented in section 6, we are confident to have provided a generic model which can directly be applied to the proofs of many more real-world protocols.

We have additionally provided a thorough discussion of the variations of our own models in section 6.3 as well as a comparison of our models with the previously existing ones in section 7. These sections should ideally help a reader in any decision with regards to the modeling of future protocols.

Some open questions have already been posed in previous sections. We will summarise here now what we consider to be the most significant ones. In general, we did not consider any formal analysis of the limitations of our model in this thesis. We consider it especially relevant to investigate whether it is actually possible to model all zero-knowledge proofs in our model. It would additionally be interesting to know what the formal differences are between our initial and the restriction-only model.

In previous models, i.e. [6] and [8], the authors outlined clear restrictions on zero-knowledge proofs such that computational soundness could still be fulfilled. At first glance, some of their arguments were not convincing. We especially were able to circumvent their condition of *extractability* (see section 7 for details) in some cases. However, as we stated in section 2.1, we did not require computational soundness in this thesis. It would be interesting however, to investigate whether or not computational soundness first of all, holds in our model and then second of all, whether it is still upheld without the *extractability* condition. An open question is indeed whether we can weaken the restrictions stipulated in [8] further or perhaps even dispose of all restrictions.

As stated in section 7, our restriction-only model seems quite close to the model from [6] which uses the notion of "true zero-knowledge proofs" in its verfication process. It is however an open question, how these two are connected in a formal sense. Also, to our knowledge, there so far exists no implementation of the afore-mentioned model. By understanding the formal connections between the two models, we would therefore also know whether our proposed model constitutes a valid implementation.

Lastly, we have not provided zero-knowledge proofs as a built-in functionality in Tamarin. By answering the above questions, one might get more clarity regarding not only the formal limitations of our model, but also whether or it could be provided as a Tamarin built-in.

References

[1] M. Chase, T. Perrin, and G. Zaverucha, "The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption," Cryptology ePrint Archive, Report 2019/1416, 2019, https://ia.cr/2019/1416.

[2] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof-Systems," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, ser. STOC '85. Association for Computing Machinery, 1985, pp. 291–304.

[3] D. Dolev and A. Yao, "On the Security of Public Key Protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983.

[4] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds. Springer, 2013, pp. 696–701.

[5] B. Blanchet and B. Smyth, "ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial," 2011.

[6] M. Backes and D. Unruh, "Computational Soundness of Symbolic Zero-Knowledge Proofs Against Active Attackers," in *2008 21st IEEE Computer Security Foundations Symposium*. IEEE, 2008, pp. 255–269.

[7] M. Backes, M. Maffei, and D. Unruh, "Zero-Knowledge in the Applied Pi-calculus and Automated Verification of the Direct Anonymous Attestation Protocol," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 202–215.

[8] M. Backes, F. Bendun, and D. Unruh, "Computational Soundness of Symbolic Zero-Knowledge Proofs: Weaker Assumptions and Mechanized Verification," in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science, D. Basin and J. C. Mitchell, Eds., vol. 7796. Springer Berlin Heidelberg, 2013, pp. 206–225.

[9] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[10] D. Jackson, "Improving automated protocol verification: Real world cryptography," PhD thesis, University of Oxford, 2020.

[11] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties," in *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 2012, pp. 78–94.

[12] R. Sasse and C. Sprenger, "Formal Methods for Information Security," Lecture notes from spring semester 2018 course, ETH Zurich.

[13] K. Milner, "Detecting the Misuse of Secrets:," PhD thesis, University of Oxford, 2018.

[14] M. Dehnel-Wild, "Component-Based Security Under Partial Compromise," PhD thesis, University of Oxford, 2018.

[15] Tamarin-Prover Manual. [Online] https://tamarin-prover.github.io/manual (Last accessed on 2021-07-18).

[16] O. Goldreich, *Foundations of Cryptography - Volume I Basic Tools*. Cambridge University Press, 2007.

[17] M. Blum, P. Feldman, and S. Micali, "Non-Interactive Zero-Knowledge and Its Applications," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC '88. Association for Computing Machinery, 1988, pp. 103–112.

[18] M. Bellare, G. Fuchsbauer, and A. Scafuro, "NIZKs with an Untrusted CRS: Security in the Face of Parameter Subversion," in *Proceedings, Part II, of the 22nd International Conference on Advances in Cryptology — ASIACRYPT 2016*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10032. Springer Berlin Heidelberg, 2016, pp. 777–804.

[19] M. Blum, A. De Santis, S. Micali, and G. Persiano, "Noninteractive Zero-Knowledge," *SIAM Journal on Computing*, vol. 20, no. 6, pp. 1084–1118, 1991.

[20] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography*. [Online] https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_5.pdf (Last accessed on 2021-08-28).

[21] M. Backes, D. Hofheinz, and D. Unruh, "CoSP: A general framework for computational soundness proofs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. Association for Computing Machinery, 2009, pp. 66–78.

[22] J. Whitefield, L. Chen, R. Sasse, S. Schneider, H. Treharne, and S. Wesemeyer, "A Symbolic Analysis of ECC-Based Direct Anonymous Attestation," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. IEEE, 2019, pp. 127–141.

[23] S. Wesemeyer, C. J. Newton, H. Treharne, L. Chen, R. Sasse, and J. Whitefield, "Formal Analysis and Implementation of a TPM 2.0-based Direct Anonymous Attestation Scheme," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. Association for Computing Machinery, 2020, pp. 784–798.

[24] W. Wang, Y. Qin, J. Liu, and D. Feng. Formal Analysis of a TTP-Free Blacklistable Anonymous Credentials System (Full Version). [Online] https://eprint.iacr.org/2017/1106.pdf (Last accessed on 2021-08-28).

[25] M. Backes, F. Bendun, M. Maffei, E. Mohammadi, and K. Pecina, "Symbolic Malleable Zero-Knowledge Proofs," in *2015 IEEE 28th Computer Security Foundations Symposium*, 2015, pp. 412–426.

[26] S. Fischlin. Tamarin theories. [Online] https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/fischlin-zk_src.zip (Last accessed on 2021-09-04).

[27] S. Fischlin. Tamarin theories, personal repository. [Online] https://github.com/inafischlin/symb-zkp.git (Last accessed on 2021-09-02).

[28] E. Brickell, J. Camenisch, and L. Chen, "Direct Anonymous Attestation," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04, no. 205. Association for Computing Machinery, 2004, pp. 132–145.