ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formalizing Aggregate Signatures in the Symbolic Model

Master Thesis

Xenia Hofmeier

25.10.2021

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

Department of Computer Science, ETH Zürich

Abstract

Security protocols are a crucial part of most applications. In recent years, automated formal verification tools, such as Tamarin and ProVerif, were effectively used to prove security properties and find attacks on complex protocols, such as 5G and TLS. To widen the range of protocols that can be modeled, we need new symbolic models of the underlying cryptographic primitives. An exciting family of signatures that have not been deeply studied in the symbolic model are multi-party signatures. This class of signatures includes blind signatures, which gained importance through e-voting, proxy signatures, which are crucial for some distributed systems, and aggregate signatures, which are applied in blockchains.

We present the first symbolic models for aggregate signatures, namely Boneh-Lynn-Shacham (BLS) signatures, in the Tamarin Prover. Signature aggregation enables combining multiple signatures into one short signature, which reduces storage and bandwidth requirements. This is especially beneficial in applications with a large number of signatures and signing parties, such as certificate chains or blockchains.

In contrast to other multi-party signature models that only consider a fixed number of parties, our model allows an arbitrary number of signers, which poses interesting challenges to correctly represent the individual elements and to achieve termination. We explore attacks on aggregate signatures and apply methods of recent works that improved symbolic models of signatures. We extend our models to cover attacks with two effective approaches: firstly, we explicitly enable certain attacks, and secondly, we create models that implicitly allow subtle behaviors that are not forbidden by the computational definition. Those methods allow us to prove properties under fewer assumptions and to find more attacks than with a standard model.

The evaluation of our models using a synthetic protocol shows that our models are effective and the analysis of isolated examples provides confidence in the correctness of our models. With our models, it is possible to develop models for real-world protocols that rely on aggregate signatures. We developed different techniques to model the aggregation of arbitrarily many signatures. Those techniques are highly promising for future models of further multi-party signatures, which would further enlarge the number of protocols that can be modeled by automated formal verification tools.

Acknowledgements

Firstly, I want to thank Dennis Jackson and Ralf Sasse for their support and guidance. I could benefit a lot from their expertise, enthusiasm for the topic, and our interesting discussions. I'm also thanking David Basin for the opportunity to write this thesis. I am thankful for the help I got from family and friends. A special thank goes to Robyn for her valuable feedback and patience.

Contents

Chapter 1

Introduction

Security protocols are omnipresent in daily life. Each visit to a webpage or usage of a mobile app includes multiple protocols. Thus, their correct design and security implications are crucial for everyone. Automated formal verification tools, such as Tamarin [22] and ProVerif [3], are powerful devices to analyze the properties of security protocols. Proofing desired properties in the development of a protocol and discovering existing attacks are essential to improve the security of systems. In recent years, Tamarin was effectively used to prove security properties and discover attacks on complex protocols, such as 5G [2] and TLS 1.3 [13].

We aim to increase the number of cryptographic primitives supported by Tamarin. This enables Tamarin to model even more protocols. Tamarin operates on the symbolic model, thus the primitives have to be abstracted. We will use Tamarin's support for user-defined functions and equational theories and some new techniques.

An exciting family of primitives that are not deeply studied in the symbolic model are multi-party signatures. This class of signatures includes blind signatures, which gained importance through e-voting, proxy signatures, which are crucial for some distributed systems [5], and aggregate signatures, which are applied in blockchains.

We present the first symbolic models for aggregate signatures, namely BLS signatures, in the Tamarin Prover. Signature aggregation was introduced by Boneh et al. [8] on BLS signatures. It enables the aggregation of multiple signatures on different messages and by different signers to one short signature. This reduces multiple signatures to a single signature, which reduces storage and bandwidth requirements. Also, for some aggregate signature schemes, such as BLS, the verification of one aggregate signature can be more efficient than verifying all aggregated signatures separately. Those properties are beneficial when a large amount of signatures is involved and the data storage, bandwidth, or computing power are limited. Boneh et al. [8] proposed using aggregate signatures for certificate chains or secure routing protocols. Another prominent application is blockchains: for example, the Ethereum 2.0 specs [1] propose the use of BLS aggregate signatures. There is an Internet

Engineering Task Force (IETF) draft for BLS signatures [9], on which some aggregate signature implementations are based [12], [21], [1].

The evaluation of our models using a synthetic protocol shows that our models are effective. The analysis of isolated examples provides confidence in the correctness of our models. With our models, it is possible to develop models for real-world protocols that rely on aggregate signatures.

1.1 Related Work

Most automated tools work in the symbolic model; manual proofs mostly use the computational model. Tamarin's cryptographic primitives are abstractions with much stronger assumptions than the primitives in the computational model. Jackson et al. [17] point out that some subtle behaviors of signatures are not captured by those symbolic models. An example of a subtle behavior is the colliding signature attack, where the adversary can create a signature that verifies for multiple messages. This attack does not violate existential unforgeability under a chosen message attack (EUF-CMA) and is thus possible in the computational model. To close this gap between the existing symbolic models and the computational model, the authors provide new models that allow such subtle behaviors and are closer to the computational definition.

They present two classes of models: the first class adds individual subtle behaviors, which enables the study of a protocol when allowing certain behaviors. This offers great help in finding and analyzing attacks and is thus referred to as attack finding models. As one cannot be sure that all possible subtle behaviors are covered, the authors introduce a second approach to modeling cryptographic primitives. Instead of explicitly allowing certain behaviors, everything that does not contradict the cryptographic definition is allowed. Thus, the model is close to the computational model. The cryptographic primitives provide weaker guarantees than in traditional symbolic models. This is especially useful to prove security properties under fewer assumptions. We thus refer to those models as validation models.

We follow this new approach to model aggregate signatures and create the same two classes of models. Our first class of models follows the standard symbolic models of signatures. Thus, to cover subtle behaviors, we have to provide the adversary with additional attack capabilities. For our second class of models, we follow the idea of Jackson et al. and derive the models from the computational definition.

The new formal verification tool Verifpal [18] supports ring signatures – a signature scheme which allows one member of a so-called ring to sign a message and the signature can be verified with the public key of any member of the ring. Verifpal supports a fixed ring size of three. In contrast, our aggregate signature models support arbitrarily many aggregated signatures. This involves exciting challenges: one is finding a suitable representation of the primitive since function symbols in Tamarin have a certain arity, and another is achieving termination the arbitrary number of aggregated signatures contributes to non-termination. This requires creative techniques. Those techniques are highly promising for future models of further multi-party signatures, which would further enlarge the number of protocols that can be modeled by automated formal verification tools.

1.2 Contributions

We develop the first practical symbolic models for aggregate signatures in Tamarin. Our models are based on the computational definition of aggregate signatures provided by Boneh et al. [8] and more specifically on BLS aggregate signatures and its IETF draft [9]. Inspired by the models of Jackson et al. [17], we develop two classes of models: attack finding models based on Tamarin's built-in signatures and validation models based on the computational definition. The development of our models includes the following contributions:

- We discuss two known attacks on aggregate signatures: rogue public key attacks and colliding signature attacks. We enable those behaviors for the attack finding models and we show that colliding signatures can occur with the validation models, without explicitly adding the behavior.
- We showcase our models on an example protocol and show that Tamarin can proof properties and find attacks for this protocol model.
- Our models are the first symbolic models of a multi-party primitive for Tamarin with arbitrarily many agents. We present different techniques that enable us to model this arbitrary number of agents.
- As the arbitrary number of aggregated signatures has a negative impact on the proof and attack finding time, we provide techniques to improve the performance.
- We justify the correctness of our models by deriving them from the computational definition and we develop a setup to evaluate the behavior of our aggregate signature models on isolated examples.

1.3 Outline

In the following chapter, we provide background on aggregate signatures and Tamarin. In Chapter 3, we give an overview of our models and describe their properties on a synthetic protocol. In the subsequent two chapters, we provide more detail on the implementation in Tamarin: in Chapter 4, we describe the attack finding models, and in Chapter 5, the validation models. In the final Chapter 6, we evaluate and compare our models. All our models are available at [15] and [16].

Chapter 2

Background

In this chapter, we provide background on aggregate signatures, specifically on Boneh-Lynn-Shacham (BLS) signatures and on the Tamarin Prover.

2.1 BLS Aggregate Signatures

In this section, we provide definitions that we will need to create our models and we give background information on signatures, and specifically aggregate signatures, BLS signatures, and specific attacks on BLS aggregate signatures. Most provided definitions are standard definitions from the textbook by Boneh and Shoup [11].

2.1.1 Signatures

Digital signatures are widely used in modern protocols, for example in certificates for public key infrastructures or software updates. They are asymmetric cryptographic primitives that can provide the authenticity and the integrity of a message. The signer uses their secret key sk to sign some message m and creates the signature $\sigma \overset{R}{\leftarrow} \mathsf{sign}(m, sk)$. Other parties, in possession of the signer's public key pk can verify the signature $\mathsf{vfy}(\sigma, m, pk)$. We use the following common definition, provided by Boneh and Shoup [11]:

Definition 2.1 (Signature scheme [11], page 529) *A signature scheme* $\mathcal{S} = (\mathsf{gen}, \mathsf{sign}, \mathsf{vfy})$ *is a triple of efficient algorithms,* gen*,* sign*, and* vfy*, where* gen *is called a key generation algorithm,* sign *is called a signing algorithm, and* vfy *is called a verification algorithm. Algorithm* sign *is used to generate signatures and algorithm* vfy *is used to verify signatures.*

- gen *is a probabilistic algorithm that takes no input. It outputs a pair* (pk, sk)*, where sk is called a secret signing key and pk is called a public verification key.*
- sign *is a probabilistic algorithm that takes as input a message* m *and a secret key sk and outputs the signature* $\sigma \overset{R}{\leftarrow} \mathsf{sign}(m, sk)$*.*

- vfy *is a deterministic algorithm invoked as* $\mathsf{vfy}(\sigma, m, pk)$ *and outputs either* true *or* false.

The ***correctness*** *property of a signature scheme requires for all key pairs* (pk, sk) *output by* gen *and all messages* m*:*

$$Pr[\mathsf{vfy}(\mathsf{sign}(m, sk), m, pk) = \mathsf{true}] = 1. \tag{2.1}$$

The messages lie in a finite message space $\mathcal{M}$ *and signatures lie in some finite signature space* Σ*. The signature scheme* $\mathcal{S}(\mathsf{gen}, \mathsf{sign}, \mathsf{vfy})$ *is defined over* $(\mathcal{M}, \Sigma)$.

The most common security definition for signatures is existential unforgeability under a chosen message attack (EUF-CMA). We define it using an attack game[1]. We provide the attack game adapted from Boneh and Shoup [11] in the next definition. We also provide the attack game in Figure 2.1.

Definition 2.2 (Attack game for EUF-CMA of signatures [11], page 530) *The attack game of a signature scheme* $\mathcal{S} = (\mathsf{gen}, \mathsf{sign}, \mathsf{vfy})$*, defined over* $(\mathcal{M}, \Sigma)$ *runs as follows between the challenger and the adversary* $\mathcal{A}$.

- *The challenger generates a key pair* (pk, sk) *with the key generation algorithm* gen*. The public key* pk *is sent to* $\mathcal{A}$.
- $\mathcal{A}$ *queries the challenger several times. For* $i = 1, 2, ...$*, the ith signing query is a message* $m_i \in \mathcal{M}$*. Given* m_i*, the challenger computes* $\sigma_i \overset{R}{\leftarrow} \mathsf{sign}(m_i, sk)$*, and then gives* σ_i *to* $\mathcal{A}$.
- *Eventually* $\mathcal{A}$ *outputs a candidate forgery pair* $(m, \sigma) \in \mathcal{M} \times \Sigma$.

The adversary wins the game, if the following two conditions hold:

- $\mathsf{vfy}(\sigma, m, pk) = \mathsf{true}$, *and*
- m *is new, it was not queried by the adversary, namey* $m \notin \{m_1, m_2, ...\}$.

We define $\mathcal{A}$*'s advantage with respect to* $\mathcal{S}$*, denoted* $SIGadv[\mathcal{A}, \mathcal{S}]$*, as the probability that* $\mathcal{A}$ *wins the game.*

On this attack game, Boneh and Shoup [11] define the security of signatures:

Definition 2.3 (Security of signatures [11], page 531) *We say that a signature scheme* $\mathcal{S}$ *is secure if for all efficient adversaries* $\mathcal{A}$*, the quantity* $SIGadv[\mathcal{A}, \mathcal{S}]$ *is negligible. Signature schemes, that are secure under this definition are existentially unforgeable under a chosen message attack.*

Note, that the key pair created by the challenger is honestly generated and the secret key is not revealed to the adversary. Thus, this security definition only implies that signatures for honestly generated and not compromised keys cannot be forged. This leads to some subtle behaviors, as discussed by Jackson et al. [17]. An example are colliding signatures, which we will discuss in Section 3.2.1. Some other limitations of this security definition are discussed by Boneh and Shoup [11] on page 532.

[1] For an explanation on attack games, see [11] Section 2.2.2

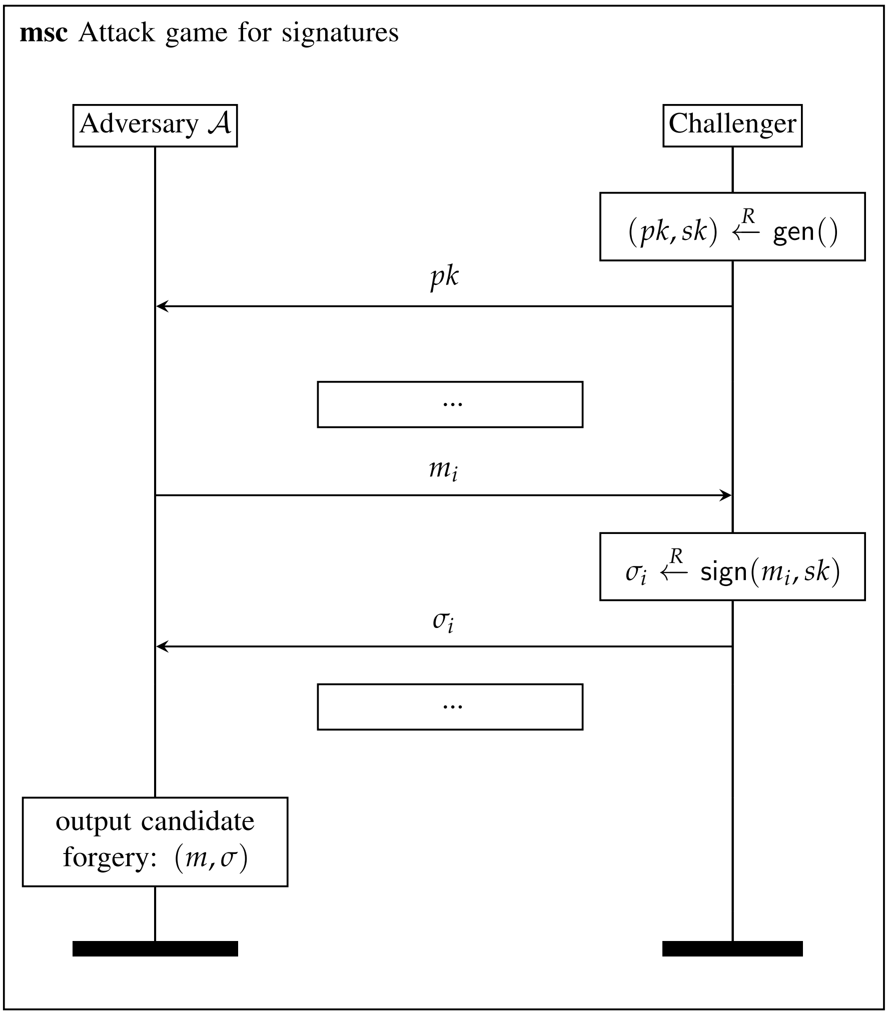
msc Attack game for signatures
Adversary $\mathcal{A}$
Challenger
$(pk, sk) \overset{R}{\leftarrow} \mathsf{gen}()$
pk
...
m_i
$\sigma_i \overset{R}{\leftarrow} \mathsf{sign}(m_i, sk)$
σ_i
...
output candidate forgery: (m, σ)

Figure 2.1: Attack game for EUF-CMA for signatures

2.1.2 Aggregate Signatures

Aggregate signature schemes, introduced by Boneh et al. [8], allow to compress multiple signatures $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$ of different messages into one short aggregate signature σ_{agg}. This compression reduces storage and bandwidth requirements. Boneh et al. propose to use aggregate signatures for certificate chains or secure routing protocols.

Boneh and Shoup [11] provide the following definition of aggregate signatures:

Definition 2.4 (Aggregate signature scheme [11], page 623) *An aggregate signature scheme* $\mathcal{SA} = (\mathsf{gen}, \mathsf{sign}, \mathsf{vfy}, \mathsf{agg}, \mathsf{vfyAgg})$ *is a signature scheme with two additional efficient algorithms* agg *and* vfyAgg*:*

- *A signature aggregation algorithm* $\mathsf{agg}(\boldsymbol{pk}, \boldsymbol{\sigma})$*: takes as input two equal length* *vectors, a vector of public keys* $\boldsymbol{pk} = (pk_1, ..., pk_n)$ *and a vector of signatures* $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$*. It outputs an aggregate signature* σ_{agg}

- *The deterministic aggregate verification algorithm* $\mathsf{vfyAgg}(\sigma_{agg}, \boldsymbol{m}, \boldsymbol{pk})$*: takes as input two equal length vectors, a vector of public keys* $\boldsymbol{pk} = (pk_1, ..., pk_n)$*, a vector of messages* $\boldsymbol{m} = (m_1, ..., m_n)$*, and an aggregate signature* σ_{agg}*. It outputs either* true *or* false.

The scheme is correct if for all $\boldsymbol{pk} = (pk_1, ..., pk_n), \boldsymbol{m} = (m_1, ..., m_n)$*, and* $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$*, if* $\mathsf{vfy}(pk_i, m_i, \sigma_i) = \mathsf{true}$ *for* $i = 1, ..., n$ *then*

$$Pr[\mathsf{vfyAgg}(\mathsf{agg}(\boldsymbol{pk}, \boldsymbol{\sigma}), \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true}] = 1 \tag{2.2}$$

Signatures can be aggregated without knowledge of the signing keys. Thus, any agent can aggregate the signatures that are in their possession.

Intuitively, the security of an aggregate signature states that an adversary that is not in possession of all of the signatures σ_1 to σ_n cannot forge the aggregate signature $\mathsf{agg}(\boldsymbol{pk}, \sigma_1, ...\sigma_n)$. We define this more formally with the attack game, provided by Boneh and Shoup [11], which is adapted from the attack game introduced by Boneh et al. [8]. Figure 2.2 illustrates the attack game.

Definition 2.5 (Attack game for EUF-CMA of agg. signatures [11], page 626) *For a given aggregate signature scheme* $\mathcal{SA} = (\mathsf{gen}, \mathsf{sign}, \mathsf{vfy}, \mathsf{agg}, \mathsf{vfyAgg})$ *with message space* $\mathcal{M}$*, and a given adversary* $\mathcal{A}$*, the attack game runs as follows:*

- *The challenger runs* $(pk, sk) \overset{R}{\leftarrow} \mathsf{gen}()$ *and sends pk to* $\mathcal{A}$*.*
- $\mathcal{A}$ *queries the challenger. For* $i = 1, 2, ...$*, the ith signing query is a message* $m^{(i)} \in \mathcal{M}$*. The challenger computes* $\sigma^{(i)} \overset{R}{\leftarrow} \mathsf{sign}(m^{(i)}, sk)$*, and then gives* $\sigma^{(i)}$ *to* $\mathcal{A}$*.*
- *Eventually* $\mathcal{A}$ *outputs a candidate aggregate forgery* $(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{agg})$ *where* $\boldsymbol{pk} = (pk_1, ..., pk_n)$ *and* $\boldsymbol{m} = (m_1, ..., m_n) \in \mathcal{M}^n$ *.*

We say that the adversary wins the game if the following conditions hold:

- $\mathsf{vfyAgg}(\sigma_{agg}, \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true}$,
- *there is at least one* $1 \leq j \leq n$ *such that (1)* $pk_j = pk$*, and (2)* $\mathcal{A}$ *did not issue a signing query for* m_j *, meaning that* $m_j \notin \{m^{(1)}, m^{(2)}, ...\}$*.*

We define $\mathcal{A}$*'s advantage with respect to* $\mathcal{SA}$*, denoted* $ASIGadv[\mathcal{A}, \mathcal{SA}]$*, as the probability that* $\mathcal{A}$ *wins the game.*

On this attack game, Boneh and Shoup [11] define the security of aggregate signatures:

Definition 2.6 (Security of aggregate signatures [11], page 626) *We say that an aggregate signature scheme* $\mathcal{SA}$ *is secure if for all efficient adversaries* $\mathcal{A}$*, the quantity* $ASIGadv[\mathcal{A}, \mathcal{SA}]$ *is negligible.*

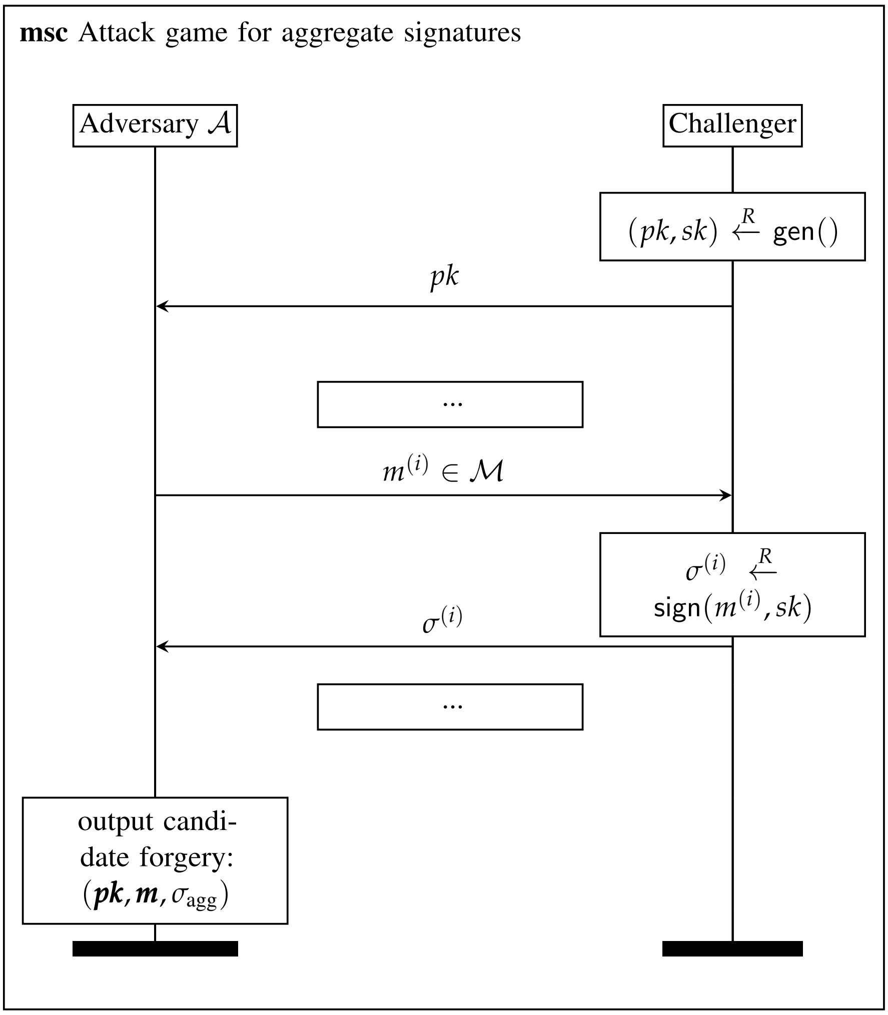
msc Attack game for aggregate signatures
Adversary $\mathcal{A}$
Challenger
$(pk, sk) \overset{R}{\leftarrow} \mathsf{gen}()$
pk
...
$m^{(i)} \in \mathcal{M}$
$\sigma^{(i)} \overset{R}{\leftarrow} \mathsf{sign}(m^{(i)}, sk)$
$\sigma^{(i)}$
...
output candidate forgery: $(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{agg}})$

Figure 2.2: Attack game for EUF-CMA for aggregate signatures

2.1.3 Bilinear Pairings

Bilinear pairings, also called bilinear maps, are definied by Boneh and Shoup [11] as follows:

Definition 2.7 (Bilinear pairing [11], page 618) *Let $\mathbb{G}_0$, $\mathbb{G}_1$, $\mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A pairing is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ satisfying the following properties:*

1. *bilinear: for all $u, u' \in \mathbb{G}_0$ and $v, v' \in \mathbb{G}_1$ we have*
   $e(u \cdot u', v) = e(u, v) \cdot e(u', v)$ and $e(u, v \cdot v') = e(u, v) \cdot e(u, v')$,
2. *non-degenerate: $g_T := e(g_0, g_1)$ is a generator of $\mathbb{G}_T$.*

If $\mathbb{G}_0 = \mathbb{G}_1$, the pairing is symmetric.

We refer to $\mathbb{G}_0$ and $\mathbb{G}_1$ as the pairing groups or source groups, and refer to $\mathbb{G}_T$ as the target group.

A central property of pairings, provided by Boneh and Shoup [11], which we will use to construct BLS signatures, is: for all $\alpha, \beta \in \mathbb{Z}_q$ we have

$$e(g_0^{\alpha}, g_1^{\beta}) = e(g_0, g_1)^{\alpha\beta} = e(g_0^{\beta}, g_1^{\alpha}) \tag{2.3}$$

2.1.4 BLS Signatures

BLS signatures were introduced by Boneh, Lynn, and Shacham [10]. The scheme is based on bilinear pairings. We here present the fomulation by Boneh and Shoup [11]:

Definition 2.8 (the BLS signature scheme [11], page 621) *Let $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ be a pairing where $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ are cyclic groups of prime order q, and where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. Let H be a hash function that maps messages in a finite set $\mathcal{M}$ to elements in $\mathbb{G}_0$.*

The BLS signature scheme, denoted $\mathcal{S}_{BLS} = (\mathsf{gen}, \mathsf{sign}, \mathsf{vfy})$, has message space $\mathcal{M}$ and works as follows:

- $\mathsf{gen}()$ *(key generation algorithm):*
  - *secret key: $sk \overset{R}{\leftarrow} \mathbb{Z}_q$*
  - *public key: $pk \leftarrow g_1^{sk} \in \mathbb{G}_1$.*
- $\mathsf{sign}(m, sk)$*: To sign a message $m \in \mathcal{M}$ using a secret key $sk \in \mathbb{Z}_q$, do $\sigma \leftarrow H(m)^{sk} \in \mathbb{G}_0$*
- $\mathsf{vfy}(\sigma, m, pk)$*: To verify a signature $\sigma \in \mathbb{G}_0$ on a message $m \in \mathcal{M}$, using the public key $pk \in \mathbb{G}_1$, output* true *if $e(H(m), pk) = e(\sigma, g_1)$*

The BLS signature scheme $\mathcal{S}_{BLS}$, is proven to be secure under the Computational Co-Diffie-Hellman (co-CDH) assumtion [10]. Boneh et al. [8] define this property as follows:

Definition 2.9 (Computational Co-Diffie-Hellman [8]) *Given $g_1, g_1^a \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$ compute $h^a \in \mathbb{G}_2$.*

Theorem 2.10 ([11], page 622) *Let $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ be a pairing and let $H : \mathcal{M} \rightarrow \mathbb{G}_0$ be a hash function. Then the derived BLS signature scheme $\mathcal{S}_{BLS}$ is a secure signature scheme assuming co-CDH holds for e, and H is modeled as a random oracle.*

For a more formal definition and proof, see [10] and [11] Section 15.5.1.

2.1.5 BLS Aggregate Signatures

BLS signatures support signature aggregation. We will derive the aggregation function by first presenting a naive approach, that is not secure.

Definition 2.11 (Naive BLS aggrgate signature [11], page 624) *The aggregate signature scheme* $\mathcal{SA}_{BLS} = (\mathcal{S}_{BLS}, \mathsf{agg}, \mathsf{vfyAgg})$*:*

- $\mathsf{agg}(\boldsymbol{pk} \in \mathbb{G}_1^n, \boldsymbol{\sigma} \in \mathbb{G}_0^n) := \{\sigma_{agg} \leftarrow \sigma_1 \cdot \sigma_2 \cdots \sigma_n \in \mathbb{G}_0,$ *output* $\sigma_{agg} \in \mathbb{G}_0\}$
- $\mathsf{vfyAgg}(\sigma_{agg}, \boldsymbol{m} \in \mathcal{M}^n, \boldsymbol{pk} \in \mathbb{G}_1^n) = \mathsf{true}$ *if*

$$e(\sigma_{agg}, g_1) = e(H(m_1), pk_1) \cdots e(H(m_n), pk_n) \tag{2.4}$$

Note that the aggregation algorithm does not require the public keys. As the aggregation does not require to validate the aggregated signatures, we will omit the public keys in our models.

The verification of BLS aggregate signatures can be optimized if all signed messages are identical $m_1 = m_2 = ... = m_n = m$. In that case, the public keys can be aggregated to one aggregate public key $pk_{agg} = pk_1 \cdots pn_n \in \mathbb{G}_1$, which is used for the verification:

$$e(\sigma_{\text{agg}}, g_1) \stackrel{?}{=} e(H(m), pk_{\text{agg}}) \tag{2.5}$$

This reduces the computation of $n + 1$ parings to two pairings. The aggregate public key can even be precomputed and reused.

2.1.6 Rogue Public Key Attack

As mentioned, the above construction is insecure since it is vulnerable to a rogue public key attack. The adversary can forge an aggregate signature that makes it seem as if a message $m \in \mathcal{M}$ was signed by a target agent whose public key is pk_{target}. The adversary creates a so-called rogue public key pk_{rogue} for a target public key pk_{target}: The adversary chooses a random value $\alpha \xleftarrow{R} \mathbb{Z}_q$ and computes:

$$pk_{rogue} \leftarrow g_1^{\alpha} / pk_{target} \in \mathbb{G}_1 \tag{2.6}$$

The corresponding secret key is $sk_{\text{rogue}} = \alpha - sk_{\text{target}}$. However, since the adversary has no access to the target secret key sk_{target}, the adversary also does not know the rogue secret key sk_{rogue}. The adversary can still create a valid, rogue aggregate signature:

$$\sigma_{\text{agg}_{\text{rogue}}} := H(m)^{\alpha} \in \mathbb{G}_0 \tag{2.7}$$

This aggregate signature is valid for twice the message m, the target public key pk_{target} and the rogue public key pk_{rogue}: $\mathsf{vfyAgg}(\sigma_{\text{agg}_{\text{rogue}}}, (m, m), (pk_{\text{target}}, pk_{\text{rogue}}))$ as:

$$\begin{aligned} e(\sigma_{\text{agg}_{\text{rogue}}}, g_1) &= e(H(m)^{\alpha}, g_1) = e(H(m), g_1^{\alpha}) \\ &= e(H(m), pk_{\text{target}} \cdot g_1^{\alpha} / pk_{\text{target}}) \\ &= e(H(m), pk_{\text{target}}) \cdot e(H(m), g_1^{\alpha} / pk_{\text{target}}) \\ &= e(H(m), pk_{\text{target}}) \cdot e(H(m), pk_{rogue}) \end{aligned}$$

The adversary creates a forgery and thus violates our security definition for aggregate signatures, Definition 2.6. Thus, we need to extend our BLS aggregate signature scheme with mitigations to prevent the rogue public key attack.

2.1.7 Mitigating the Rogue Public Key Attack

There are different methods to prevent the rogue public key attack. We present three of them here.

Prevent Duplicate Messages

Note that the rogue public key aggregate $\sigma_{\text{agg}_{\text{rogue}}}$ validates for twice the message m. In fact, this attack is only possible with multiple times the same message. The first mitigation, presented by Boneh et al. [8], addresses this, by enforcing distinct messages. The verification algorithm verifies the distinctness of the messages and otherwise rejects. This is only suitable for applications with unique messages, for example certificate chains.

Message Augmentation

The second method can be applied, if distinct messages are not given. In that case, the distinctness is achieved by prepending the signing public key to every message, before signing.

Definition 2.12 (BLS agg. signature with message augmentation [11], page 626) *Our modified aggregation scheme, denoted $\mathcal{SA}^{(1)}_{BLS}$, is the same as $\mathcal{SA}_{BLS}$ in 2.11 except that the signing algorithm now uses a hash function $H : G_1 \times \mathcal{M} \to \mathbb{G}_0$ and is defined as*

$$\mathsf{sign}(m, sk) := H(pk, m)^{\alpha} \textit{ where } sk = \alpha \in \mathbb{Z}_q \textit{ and } pk \to g_1^{\alpha} \tag{2.8}$$

In effect, the message being signed is the pair $(pk, m) \in \mathbb{G}_1 \times \mathcal{M}$. The verification and aggregate verification algorithms are equally modified to hash the pairs (pk, m). Specifically, aggregate verification works as

$$\begin{aligned} &\mathsf{vfyAgg}(\sigma_{agg}, \boldsymbol{m} \in \mathcal{M}^n, \boldsymbol{pk} \in \mathbb{G}_1^n) = \mathsf{true} \\ &\qquad \textit{if } e(\sigma_{agg}, g_1) = e(H(pk_1, m_1), pk_1) \cdots e(H(pk_n, m_n), pk_n) \end{aligned} \tag{2.9}$$

As described in Section 2.1.5, the naive approach for BLS aggregate signatures of Definition 2.11 can be verified faster, if all messages are the same. This optimization is not possible with this method.

Proof of Possession of the Secret Key

This method preserves the fast verification described in Section 2.1.5. Recall that in the rogue public key attack, the adversary is not in possession of the rogue secret key pk_{rogue}. We use this fact for this mitigation approach: The signers have to prove the possession of their secret keys, referred to as Proof of Possession (PoP).

Definition 2.13 (BLS aggregate signature with PoP [11], page 627) *The modified aggregation scheme, denoted $\mathcal{SA}^{(2)}_{BLS}$, is the same as $\mathcal{SA}_{BLS}$ defined in 2.11 except that the key generation algorithm also generates a proof π to show that the signer has possession of the secret key. We attach this proof π to the public key, and it is checked during aggregate verification. In particular, the key generation and aggregate verification algorithms use an auxiliary hash function $H' : \mathbb{G}_1 \to \mathbb{G}_0$, and operate as follows:*

- $\mathsf{gen}() := \left\{ \begin{array}{c} \alpha \xleftarrow{R} \mathbb{Z}_q, u \leftarrow g_1^{\alpha} \in \mathbb{G}_1, \pi \leftarrow H'(u)^{\alpha} \in \mathbb{G}_0 \\ \textit{output } pk := (u, \pi) \in \mathbb{G}_1 \times \mathbb{G}_0 \textit{ and } sk := \alpha \in \mathbb{Z}_q \end{array} \right\}.$
- $\mathsf{vfyAgg}(\sigma_{agg}, \boldsymbol{m} \in \mathcal{M}^n, \boldsymbol{pk})$:
  Let $\boldsymbol{pk} = (pk_1, ..., pk_n) = ((u_1, \pi_1), ..., (u_n, \pi_n))$ be n public keys, and let $\boldsymbol{m} = (m_1, ..., m_n)$. Accept if
  - *valid proofs: $e(\pi_i, g_1) = e(H'(u_i), u_i)$ for all $i = 1, ..., n$, and*
  - *valid aggregate: $e(\sigma_{agg}, g_1) = e(H(m_1), u_1) \cdots e(H(m_n), u_n)$*

The new term $\pi = H'(u)^{\alpha}$ in the public key is used to prove that the public key owner is in possession of the secret key α. This π is a BLS signature on the public key $u \in \mathbb{G}_1$, but using the hash function H' instead of H. The aggregate verification algorithm first checks that all the terms $\pi_1, ..., \pi_n \in \mathbb{G}_0$ in the given public keys are valid, and then verifies that the aggregate signature σ_{agg} is valid exactly as in $\mathcal{SA}_{BLS}$.

The above presented schemes $\mathcal{SA}^{(1)}_{BLS}$ and $\mathcal{SA}^{(2)}_{BLS}$ are secure in the sense of Definition 2.6, assuming co-CDH holds for e, and the Hash functions H and H' are modeled as random oracles, see [7] and [11] Section 15.5.3.3 for the proofs.

Aggregate signatures and multi-signatures are sometimes confused, also because there are multi-signature schemes for BLS signatures, for example the schemes from Boldyreva [4] or Boneh et al. [6]. Aggregate signatures and multi-signatures both reduce multiple signatures to a single short signature. The main difference between the two is that the signatures of a multi-signature have to be on the same message where aggregate signatures allow signatures on different messages. Multi-signatures also contain some more elements such as a system parameter. The multi-signature is created collectively between the parties and the signing algorithm is interactive.

2.1.8 Implementations

There exists a Internet Engineering Task Force (IETF) draft of BLS aggregate signatures [9]. The document is work in progress. It documents the three rogue public key mitigations that we mentioned above in form of three schemes. The PoP scheme supports fast verification, as described in Section 2.1.5 on page 11. As mentioned above, the BLS aggregation algorithm does not require the public keys, in contrast to the definition of aggregate signatures 2.4 which includes the public keys. The IETF draft also defines the arguments without the public keys:

```
signature = Aggregate((signature_1, ..., signature_n))
```

There are several implementations of this IETF draft, mostly related to blockchains. Here are three examples:

- Chia, a blockchain company, provides an implementation [12], which is not yet reviewed.
- The blockchain company Algorand has a work in progress BLS signature implementation [21].
- The Ethereum 2.0's beacon chain specification [1] includes BLS aggregate signatures.

2.1.9 Attacks on BLS Signatures

Quan [19], [20] describes several attacks on BLS aggregate signatures, with a focus on the IETF draft [9]. In this section, we describe three of those attacks. We evaluate the second attack with our models.

Identity Element Keys with a Single BLS Signature

Quan [19] describes an attack on BLS signatures. The adversary can choose a malicious signing key, which results in signatures that are accepted for all messages.

The adversary chooses the private key to be equal to zero, this results in the corresponding pubic key and all signatures signed with it being the identity element:

$$sk_{\text{adv}} := 0 \tag{2.10}$$

$$pk_{\text{adv}} := g_1^{sk_{\text{adv}}} = g_1^0 = 1 \tag{2.11}$$

$$\sigma_{\text{adv}} := H(m)^{sk_{\text{adv}}} = H(m)^0 = 1 \tag{2.12}$$

As $\mathsf{vfy}(\sigma_{\text{adv}}, m, pk_{\text{adv}}) = \mathsf{true}$ if $e(H(m), pk_{\text{adv}}) = e(\sigma_{\text{adv}}, g_1)$, and $e(H(m), 1) = e(1, g_1)$ for any message $m \in \mathcal{M}$, the signature σ_{adv} validates for the public key pk_{adv} and any message $m \in \mathcal{M}$.

This attack does not violate the security definition of signatures 2.3, as the signing key sk_{adv} is not honestly generated. This attack is an instance of a colliding signature attack, which we will discuss in Section 3.2.1.

The BLS IETF draft [9] requires in the verification algorithm to check that the public key is not the identity element. According to Quan [19], some libraries implementing the IETF draft, omit this check or implement it incorrectly, which makes this attack practical for those implementations.

Splitting Zero Attack

Quan [19] extends the above attack for BLS aggregate signatures. The splitting zero attack bypasses the identity element check, by using multiple malicious keys.

The adversary chooses two malicious keys, such that $sk_{\text{mal}_1} + sk_{\text{mal}_2} = 0$. And both sk_{mal_1} and sk_{mal_2} are non-zero. Thus, the corresponding public keys are not the identity element and the public keys will validate. The adversary then signs some message m with each malicious key sk_{mal_1} and sk_{mal_2}. This results in the aggregate of those signatures being the identity element:

$$\begin{aligned}\sigma_{\text{agg}_{\text{mal}}} &:= \sigma_{\text{mal}_1} \cdot \sigma_{\text{mal}_2} \\ &= H(m)^{sk_{\text{mal}_1}} \cdot H(m)^{sk_{\text{mal}_2}} \\ &= H(m)^{sk_{\text{mal}_1} + sk_{\text{mal}_2}} \\ &= H(m)^0 \\ &= 1\end{aligned}$$

Note that the product of the two malicious public keys is the identity element:

$$pk_{\text{mal}_1} \cdot pk_{\text{mal}_2} = 1 \tag{2.13}$$

The adversary can aggregate this malicious signature $\sigma_{\text{agg}_{\text{mal}}} = 1$ with some third valid signature σ_3. This results in a aggregate signature equal to this third signature σ_3:

$$\sigma_{\text{agg}_{1,2,3}} := \sigma_{\text{mal}_1} \cdot \sigma_{\text{mal}_2} \cdot \sigma_3 = \sigma_3 \tag{2.14}$$

This aggregate signature will validate against the messages vector (m', m', m_3) where m' can be any message. We show this in the following: As σ_3 is a valid signature, we have $e(g_1, \sigma_3) = e(pk_3, H(m_3))$. The following derivation shows, that $\mathsf{vfyAgg}(\sigma_{\text{agg}_{1,2,3}}, (m', m', m_3), (pk_{\text{mal}_1}, pk_{\text{mal}_2}, pk_3)) = \mathsf{true}$ for any message m':

$$\begin{aligned}e(g_1, \sigma_{\text{agg}_{1,2,3}}) &= e(g_1, \sigma_3) \\ &= 1 \cdot e(pk_3, H(m_3)) \\ &= e(g_1, H(m'))^0 \cdot e(pk_3, H(m_3)) \\ &= e(g_1, H(m'))^{sk_{\text{mal}_1} + sk_{\text{mal}_2}} \cdot e(pk_3, H(m_3)) \\ &= e(g_1, H(m'))^{sk_{\text{mal}_1}} \cdot e(g_1, H(m'))^{sk_{\text{mal}_2}} \cdot e(pk_3, H(m_3)) \\ &= e(g_1^{sk_{\text{mal}_1}}, H(m')) \cdot e(g_1^{sk_{\text{mal}_2}}, H(m')) \cdot e(pk_3, H(m_3)) \\ &= e(pk_{\text{mal}_1}, H(m')) \cdot e(pk_{\text{mal}_2}, H(m')) \cdot e(pk_3, H(m_3))\end{aligned}$$

As the above zero bug, this attack does not violate the security definition of aggregate signatures 2.6. It is still an unexpected behavior of aggregate signatures. One might expect, that if two agents are in possession of the same aggregate signature, they will validate it with the same messages. However, this cannot be expected if the scheme only guarantees EUF-CMA.

This attack may be a problem if one expects a signature to provide consensus. But this is not the case for aggregate signatures nor signatures. There are no guarantees for malicious keys. Other primitives or protocols have to provide consensus, this cannot be achieved by signatures.

Interestingly, the first two methods to prevent the Rogue public key attack (preventing duplicate messages and message augmentation), will also prevent this attack, since aggregating signatures on the same message will be prevented. But the PoP method does not resolve this problem, as the adversary is in possession of the two malicious keys.

As each subset of public keys with the same signed message could be colluded, the attack could only be prevented, by checking each such subset of public keys. This would be quite expensive.

Splitting Zero Attack against FastAggregateVerify

As described in Section 2.1.5, BLS aggregate signatures enable a faster verification if all signatures are equal. This option is also included in the IETF draft [9]. For the PoP scheme, there are two algorithms to verify an aggregate signature: `AggregateVerify` and `FastAggregateVerify`. The first one performs a normal vfyAgg and validates every public key individually, where the second algorithm performs the optimization and validates the aggregate of the public keys. Thus, in the following example, the two algorithms will return different results for the same input:

Example 2.14 *We validate a malicious aggregate signature, as in the above splitting zero attack:* $\sigma_{agg_{mal}} := 1$ *with two malicious public keys with a product equal to the identity element:* $pk_{mal_1} \cdot pk_{mal_2} = 1$*. The two verification algorithms will now provide different results:*

$$\texttt{AggregateVerify}((m,m),(pk_{mal_1},pk_{mal_2}),0) = \text{true} \tag{2.15}$$

$$\texttt{FastAggregateVerify}(m,(pk_{mal_1},pk_{mal_2}),0) = \text{false} \tag{2.16}$$

As mentioned above, `AggregateVerify` *will validate the two public keys* pk_{mal_1} *and* pk_{mal_2} *separately. As they are both not the identity element, the algorithm returns* true. `FastAggregateVerify` *aggregates the two public keys by multiplying them* $pk_{mal_1} \cdot pk_{mal_2} = 1$*. The validation of this aggregated public key will fail, as it is equal to the identity element.*

This could be prevented, by additionally validating the aggregated key in the algorithm AggregateVerify if all messages are the same. This would lead to some computational overhead, which may not be desirable.

2.2 Tamarin

The Tamarin Prover [22] is an automated formal verification tool to model and analyze security protocols. When providing a symbolic model of the protocol and the desired security properties to Tamarin, it can construct a proof or an attack on the property, or it does not terminate due to the undecidability of the problem.

As Tamarin operates in the symbolic model, cryptographic messages are modeled as terms. Each cryptographic message is a constant, a variable, or a function symbol applied to some messages. The properties of functions are modeled using equational theories, which are sets of equations. To illustrate this, let us look exemplary at the function symbol pair. It models a pair of two messages. To access the first and second argument, we use the two additional function symbols fst and snd. Tamarin supports the following two equations:

```
fst(pair(x,y)) = x
snd(pair(x,y)) = y
```

In this thesis, we will write <x, y> for pair(x, y), which is supported by Tamarin as syntactic sugar. Tamarin supports various equational theories for cryptographic primitives. For example, Tamarin's built-in equational theory for signatures is defined as follows:

```
functions: sign/2, verify/3, pk/1, true/0
equations: verify(sign(m,sk), m, pk(sk)) = true
```

The verification returns true if the message and secret key inside the signature function sign match the message and public key provided for the verification, stated as the second and third argument of verify. As described in Section 1.1, Jackson et al. [17] discussed that this standard model does not capture some subtle behaviors, such as colliding signatures, which we will discuss in Section 3.2.1. Functions and equations can also be defined by the user. We will use this functionality and some other mechanism to model our aggregate signatures.

The protocols are modeled using labeled multiset-rewrite rules, which define a labeled transition system. The current protocol state is represented by a multiset of facts, which can transition into a new state through the application of a rule. Let us look at the following example rule and discuss its components:

```
rule Example:
  [ Fr(~m) ]
--[ Label(~m) ]->
  [ State('1', ~m), Out(pair(~m, ~m)) ]
```

A multiset-rewrite rule in Tamarin consists of a name, here `Example`, a left hand side on line 2, a right hand side on line 4, which are both multisets of facts, and a multiset of labels on line 3, called action facts.

A rule can be applied if there is a subset of facts in the current state that matches the left hand side of the rule. Applying the rule results in the removal of this matching subset of facts and the addition of instantiations of the facts in the right hand side with the matching substitution.

Note that the above rule contains two special facts: `Fr` and `Out`. `Fr` is used to access so-called fresh variables, marked with the prefix ~. Tamarin has built-in rules that generate instances of `Fr(x)` and Tamarin ensures that each term instantiating `x` is unique. In the above example rule, we have the fresh variable `~m` and the public constant `'1'`, which is a publicly known atomic message. The `Out` fact, together with the `In` fact are used to model the Dolev-Yao adversary, which represents the untrusted network. `Out` facts are used to model agents sending messages, in the above example `pair(~m, ~m)`, to the network and `In` facts for receiving messages from the network.

The execution of a protocol is represented by the repeated application of multiset rewrite rules to the protocol's state, where the first state is an empty multiset. The trace of such an execution is defined by the sequence of the instantiated action facts of the rewrite rules. For example, if a protocol execution consisted of the application of the following two rules consecutively,

```
rule FirstRule:
  [ ]
--[ L1() ]->
  [ F('1') ]

rule SecondRule:
  [ F(a) ]
--[ L2(a), L3() ]->
  [ G('2') ]
```

the trace of this execution would be {`L1()`},{`L2('1')`,`L3()`}. Note that the trace contains the action fact `L2('1')` while the second rule contains `L2(a)`. The variable `a` is instantiated by the constant `'1'`. The rules could of course also be applied in a different order, which would result in different traces, for example: {`L1()`},{`L1()`},{`L1()`}. We define the semantics of a security protocol as the set of all traces of the protocol's labeled transition system. We express the security properties as trace properties, which are also sets of traces. This means a protocol satisfies a security property if the protocol's set of traces is a subset of the propriety's traces. If this is not the case, attacks on the protocol that violate the property are possible. Those attacks are represented as traces that are not in the property's set of traces.

In Tamarin, we formulate the security properties as first order logic formulas, so-called lemmas. There are two kinds of lemmas: the most common one express that the property must hold for all traces of the protocol, while the executability lemmas, marked with the keyword `exists-trace`, state that there exists a trace for which the property holds. The first kind of lemmas is used to express, that a property holds for a protocol, while the second kind of lemmas is mostly used to check that the protocol model is executable.

Chapter 3

Overview

This overview presents an introduction to our aggregate signature models[1]. We first introduce a motivating protocol and then discuss our two families of models in detail. In a similar vein as Jackson et al. [17], we created two classes of aggregate signature models: one for attack finding and one for validation. The attack finding models are built around equational theories, whereas the validation models rely on Tamarin's support for restrictions.

3.1 Motivating Example Protocol

We first provide a motivating example for aggregate signatures. We look at a synthetic protocol to crowdsource weather data. Figure 3.1 illustrates this protocol. In a meteorological project, private weather stations share their measurements with the public. We assume the weather stations do not have access to the internet; they have however, some means of wireless broadcasting. Volunteers in the role of the aggregator collect the measurements of multiple stations. They provide the collected data to the public. To ensure the authenticity of the data, the weather stations sign their measurements and broadcast the signatures along with the measurements. To save bandwidth, the volunteers aggregate the signatures. The verifiers access the published data and confirm its authenticity by validating the aggregate signatures.

The message sequence chart in Figure 3.2 shows the protocol in more detail.

1. The weather stations S_1 to S_n each send out their measurement m_i, the corresponding signature $\sigma_i = \mathsf{sign}(m_i, sk_i)$ and their agent name S_i.
2. The aggregator receives them and publishes all the collected measurements along with the stations' names and the aggregated signatures $\mathsf{agg}(\sigma_1, ..., \sigma_n)$.
3. The verifier first looks up the keys with the station names and then verifies the aggregation.

[1] All our models are available at [15] and [16]

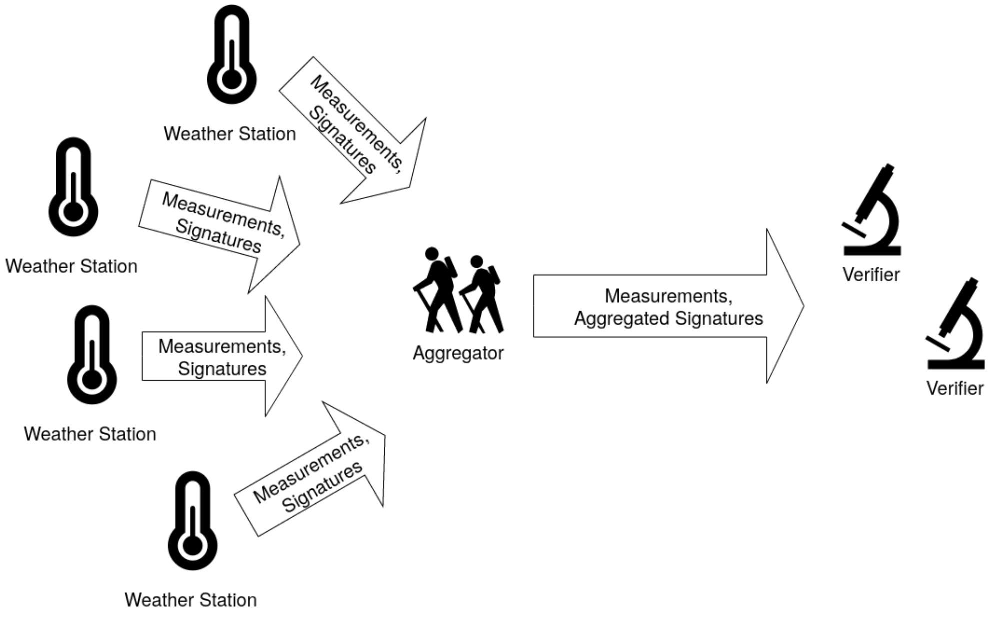

Figure 3.1: Visualization of the crowdsourced weather protocol

3.1.1 Protocol Properties

To characterize this protocol, we formulate two authentication properties, namely message authenticity and weak agreement.

The message authenticity property states that when the verifier V has received a message m from a station S, the station S has previously sent out that message m or one of the agents V or S is not honest. This is an extension of the well-known aliveness property with an additional agreement on the message. We define this property more formally as Lemma 6.1 on page 49.

From the security definition of aggregate signatures (see Definition 2.6 on page 8), we know that for an honestly generated public key pk_i and a message m_i for which an aggregate signature σ_{agg} validates

$$\mathsf{vfyAgg}(\sigma_{\text{agg}}, (m_1, ..., m_i, ..., m_n), (pk_1, ..., pk_i, ..., pk_n)) = \mathsf{true} \tag{3.1}$$

the agent owning the corresponding secret key sk_i must have created the signature $\sigma_i = \mathsf{sign}(m_i, sk_i)$. Thus the message authenticity property holds.

The well-known weak agreement property states that when the verifier V has run the protocol apparently with the station S then the station S must have run the protocol apparently with V. This is defined more formally as Lemma 6.2 on page 49.

As the station S does not include the intended destination, any verifier will accept the message. And thus, weak agreement does not hold.

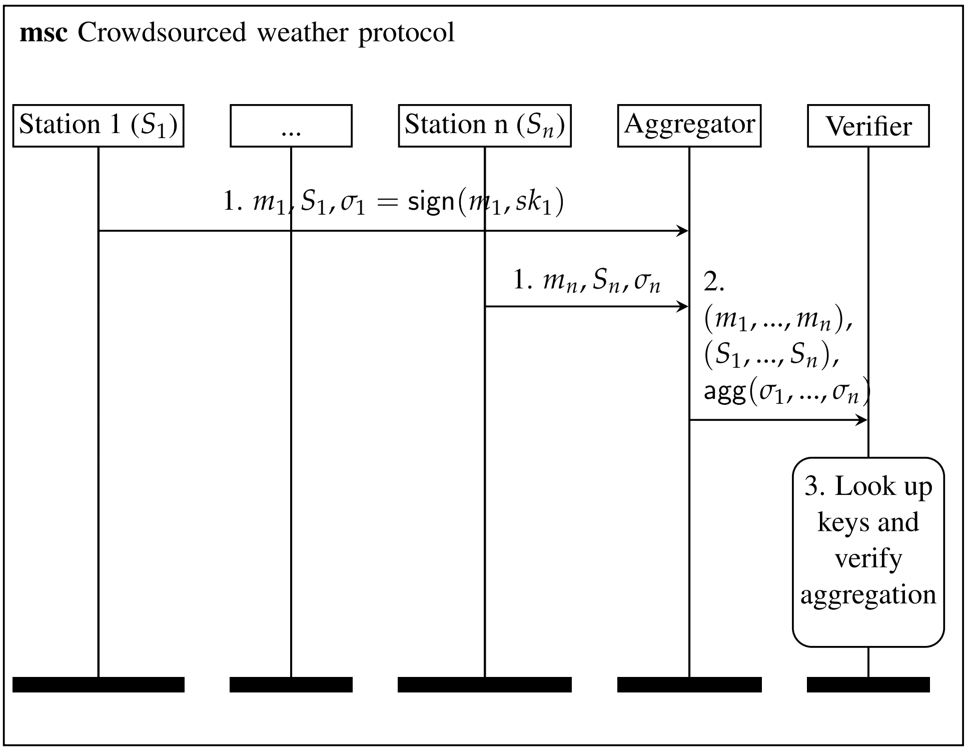
msc Crowdsourced weather protocol
Station 1 (S_1)
...
Station n (S_n)
Aggregator
Verifier
1. $m_1, S_1, \sigma_1 = \mathsf{sign}(m_1, sk_1)$
1. m_n, S_n, σ_n
2. $(m_1, ..., m_n)$, $(S_1, ..., S_n)$, $\mathsf{agg}(\sigma_1, ..., \sigma_n)$
3. Look up keys and verify aggregation

Figure 3.2: Message sequence chart of the crowdsourced weather data protocol

3.2 Attack Finding Models

The aggregate verification function vfyAgg is defined over whether the aggregated signatures validate or not. Thus, we have to choose a signature model with which we design the aggregate signature model. In a first step, we choose Tamarin's built-in signatures: A signature σ, is accepted ($\mathtt{vfy}(\sigma, m, \mathtt{pk}(sk)) = \mathtt{true}$), exactly if $\sigma = \mathtt{sign}(m, sk)$ where $\mathtt{pk}(sk)$ is the public key corresponding to sk. This is independent on whether the key is honestly generated or not.

The definition of aggregate signatures (see Definition 2.4 on page 7) defines the aggregation function to have as input a list of public keys and a list of signatures. However, the aggregation of BLS aggregate signatures (see Definition 2.11 on page 11) only multiplies the signatures and omits the public keys. Also, the IETF draft of BLS [9] omits the public keys in the aggregation algorithm. Thus we will only include the signatures in the aggregation.

Based on the above, we define:

Definition 3.1 (Aggregate signatures based on Tamarin's built-in signatures)

$$\mathtt{vfyAgg}(\sigma_{agg}, (m_1, ..., m_n), (\mathtt{pk}(sk_1), ..., \mathtt{pk}(sk_n))) = \mathsf{true} \tag{3.2}$$

exactly if $\sigma_{agg} = \mathsf{agg}(\sigma_1, ..., \sigma_n)$ *where* $\sigma_i = \mathtt{sign}(m_i, sk_i)$ *for all* $i = 1, ..., n$.

Our attack finding aggregate signature model is based on this definition. We model our weather protocol with this aggregate signature model. It behaves as expected: we can prove message authenticity and disprove the weak agreement property.

3.2.1 Colliding Signatures

Our model so far does not differentiate between honestly generated and maliciously created keys. In practice, non-honestly generated keys can lead to severe attacks. Jackson et al. [17] discuss different attacks on signatures that are possible due to non-honest keys. One example is the colliding signature attack:

Definition 3.2 (Colliding signatures) *The adversary constructs a key pair sk_{mal}, pk_{mal} and a signature σ_{mal} such that the signature will validate against multiple messages:* $\mathsf{vfy}(\sigma_{mal}, m, pk_{mal}) = \mathsf{vfy}(\sigma_{mal}, m', pk_{mal}) = \mathsf{true}$ *where $m \neq m'$.*

Note that this attack does not violate the security definition of signatures: in the attack game for signatures (see Definition 2.2 on page 6), the challenger generates the key pair honestly and the adversary does not learn the secret key. Thus, the security of signatures only gives guarantees for honestly generated keys.

So far, we have assumed that all weather stations are honest. In a small meteorology association, this might be realistic. But what if the project is open to everyone? What if some weather station owners have malicious intentions? For example, a hotel owner reports good weather to attract tourists. Of course, in our protocol, the hotel owner could send any message as weather measurement. In this case, every verifier would work with the same incorrect data. But what if the hotel owner could, for example, make sailors believe that there is a lot of wind and sunbathers that the wind is calm?

For a single signature, this is possible with a colliding signature attack. For BLS signatures, a secret key equal to zero can be used as a malicious key. But this would lead to the public key being the identity element, which can be detected easily. The IETF draft of BLS [9] requires validation that the public keys are not the identity element. For BLS aggregate signatures, Quan [19] describes the splitting zero attack which is more difficult to detect.

We describe the splitting zero attack in detail in Section 2.1.9. Here we provide a brief abstract. The adversary constructs colluded malicious keys $sk_{\mathrm{mal}_1}, ..., sk_{\mathrm{mal}_k}$ to create signatures on an arbitrary message m. These malicious signatures are aggregated possibly with some honest signatures $(\sigma_l, ..., \sigma_n)$:

$$\sigma_{\mathrm{agg}_{\mathrm{mal}}} = \mathsf{agg}(\mathsf{sign}(m, sk_{\mathrm{mal}_1}), ..., \mathsf{sign}(m, sk_{\mathrm{mal}_k}), \sigma_l, ..., \sigma_n) \tag{3.3}$$

In the verification step, the message m can be replaced by any message m'.

$$\begin{aligned}&\mathsf{vfyAgg}(\sigma_{\mathrm{agg}_{\mathrm{mal}}}, (m, ..., m, m_l, ..., m_n), (pk_{\mathrm{mal}_1}, ..., pk_{\mathrm{mal}_k}, pk_l, ..., pk_n))\\ =&\mathsf{vfyAgg}(\sigma_{\mathrm{agg}_{\mathrm{mal}}}, (m', ..., m', m_l, ..., m_n), (pk_{\mathrm{mal}_1}, ..., pk_{\mathrm{mal}_k}, pk_l, .., pk_n))\\ =&\mathsf{true}\end{aligned}$$

Note that only the messages signed by the malicious keys (marked in red) can be replaced. The messages m_l to m_n signed by honest keys cannot be replaced. Following the same argumentation as for the colliding signature attack, this attack does not violate the security definition of aggregate signatures.

The IETF draft of BLS [9] does not specify measures against splitting zero attacks. As explained in Section 2.1.9 on page 16, it would be computationally expensive to detect such attacks. Thus, this is a realistic attack and our model should account for it.

In the splitting zero attack, all the messages signed by the colluded malicious keys must be the same. But according to the security definition, each message signed by a malicious key could be replaced separately. We model this more general colliding signature attack. We define the additional colliding signature behavior as follows: For some malicious keys $sk_{\mathrm{mal}_1}, ...sk_{\mathrm{mal}_k}$, messages $m_1, ..., m_k, m_l, ..., m_n$, honest keys $sk_l, ..., sk_n$, and honest signatures $\sigma_l = \mathtt{sign}(m_l, sk_l), ..., \sigma_n = \mathtt{sign}(m_n, sk_n)$, the aggregate signature

$$\sigma_{\mathrm{agg}_{\mathrm{mal}}} = \mathtt{agg}(\mathtt{sign}(m_1, sk_{\mathrm{mal}_1}), ..., \mathtt{sign}(m_k, sk_{\mathrm{mal}_k}), \sigma_l, ..., \sigma_n) \tag{3.4}$$

will validate against any messages $m'_1, ..., m'_k$ and the messages $m_l, ..., m_n$:

$$\begin{aligned}&\mathtt{vfyAgg}(\sigma_{\mathrm{agg}_{\mathrm{mal}}}, (m'_1, ..., m'_k, m_l, ..., m_n),\\ &\qquad (pk_{\mathrm{mal}_1}, ..., pk_{\mathrm{mal}_k}, \mathtt{pk}(sk_l), .., \mathtt{pk}(sk_n))) = \mathsf{true}\end{aligned} \tag{3.5}$$

With this extended model for finding colliding signature attacks, we can produce an attack trace that corresponds to the splitting zero attack described by Quan [19]. The attack is depicted in Figure 3.3. The adversary aggregates an honest signature σ and a malicious signature $\mathtt{sign}(m_a, sk_{\mathrm{mal}})$. The adversary sends this aggregate signature to two verifiers but with different messages m_b and m_c. Both verifiers validate σ_{agg} successfully, although they did not use the same messages for the verification. Note that the message signed by the station cannot be exchanged. Thus, the colliding signature attack is not an attack on our message authenticity property as the property must only hold for honest agents. Our colliding signature model is still able to prove message authenticity and disprove weak agreement.

Let us have another look at our malicious hotelier. The colliding signature attack enables reporting wind to some verifiers and no wind to others, although everyone is in possession of the same aggregate signature. This points out that the possession of the same signature (aggregate or not) does not provide consensus.

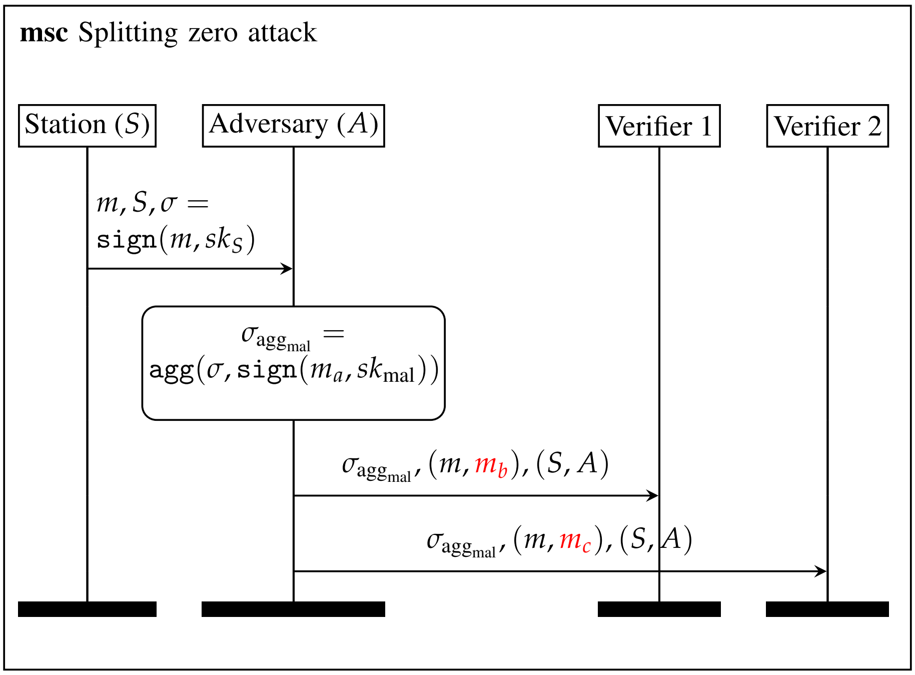
msc Splitting zero attack
Station (S)
Adversary (A)
Verifier 1
Verifier 2
$m, S, \sigma = \mathtt{sign}(m, sk_S)$
$\sigma_{\mathrm{agg}_{\mathrm{mal}}} = \mathtt{agg}(\sigma, \mathtt{sign}(m_a, sk_{\mathrm{mal}}))$
$\sigma_{\mathrm{agg}_{\mathrm{mal}}}, (m, m_b), (S, A)$
$\sigma_{\mathrm{agg}_{\mathrm{mal}}}, (m, m_c), (S, A)$

Figure 3.3: Attack trace of a splitting zero attack with one honest sk_S and one malicious key sk_{mal}.

3.2.2 Rogue Public Key Attack

The rogue public key attack, described in detail in Section 2.1.6, enables an adversary to forge an aggregate signature. The adversary creates a rogue public key for a target public key and a rogue aggregate signature that validates for this rogue public key and the target public key. When Boneh et al. [8] introduced BLS aggregate signatures, they pointed out that one has to prevent rogue public key attacks. But some implementations might still miss the mitigations or implement them incorrectly. Thus, rogue public key attacks might still be possible in practice. Therefore, we want to be able to explore what consequences a rogue key attack has on a protocol. We provide an attack finding model which enables the adversary to perform a rogue public key attack.

We introduce two new functions: `roguePk` enables the adversary to create a rogue public key for a target public key pk_{target}:

$$pk_{\mathrm{rogue}} = \mathtt{roguePk}(pk_{\mathrm{target}}) \tag{3.6}$$

and `rogueAgg` enables the adversary to create a rogue aggregate signature for a target message m and rogue public key pk_{rogue}:

$$\sigma_{\mathrm{agg}_{\mathrm{rogue}}} = \mathtt{rogueAgg}(m, pk_{\mathrm{rogue}}) \tag{3.7}$$

This rogue aggregate signature validates for twice the message m, the rogue public key pk_{rogue}, and the victim's public key pk_{target}:

$$\texttt{vfyAgg}(\sigma_{\text{agg}_{\text{rogue}}}, (m, m), (pk_{\text{rogue}}, pk_{\text{target}})) = \texttt{true} \tag{3.8}$$

Note that the possession of the public key is sufficient for the attack. The adversary does not need a valid signature nor the victim's secret key. The forged signature lets a verifier believe that the station owning sk_{target} signed the message m. Thus, the message authenticity property no longer holds. With this additional attack, we have almost no security guarantees left, but a colliding signature attack is here not possible. A rogue aggregate signature will not validate against different messages.

The malicious hotelier could perform a rogue public key attack on a competing hotel by reporting bad weather at that hotel. In contrast to the colliding signature attack, all verifiers in possession of the rogue signature will have to validate with the same message.

3.3 Validation Model

We have seen in Section 3.2.1 that the splitting zero attack does not violate the security definition but it is not covered by a standard model based on Tamarin's built-in signatures. Variants of the subtle attacks described by Jackson et al. [17] for signatures, as described in Section 1.1, may also be possible for aggregate signatures. We can add more attacks to the attack finding model, but there is no guarantee that we would cover all possible subtle behaviors. Jackson et al. solve this problem by introducing a validation model. It is based on the computational definition of signatures. Each subtle behavior that is not ruled out by the computational definition of signatures is allowed. We follow this idea and create a validation model for aggregate signatures.

Our validation model relies on the Correctness Definition 2.4 and the EUF-CMA security Definition 2.6. The correctness of aggregate signatures states that if all signatures σ_i in an aggregation $\mathsf{agg}(\sigma_1, ..., \sigma_n)$ are valid and their signing keys are honest, the aggregation has to be valid as well. Existential unforgeability states that if an aggregate signature is accepted by the verification algorithm ($\mathsf{vfyAgg}(\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true}$), all aggregated signatures σ_i are either valid or the public key pk_i is non-honest. Note that the correctness and security definition describe the result of verifications for honest keys. There is no statement for non-honest keys. Thus a verification including a non-honest key could be true or false; however, as the verification algorithm vfyAgg is deterministic, the result of a verification always has to be the same. We call this property consistency. We model the verification of aggregate signatures with those three properties: correctness, security and consistency. We define them more formally in Section 5.1 on page 38 and express them in Tamarin as restrictions (see Restrictions 5.1, 5.3, 5.4).

When disallowing malicious keys, this validation model behaves similarly to the attack finding model without additional adversary capabilities. When adding the ability to register malicious keys, our Tamarin theory finds the splitting zero attack. In contrast to the attack finding models, we do not need to explicitly allow attacks, but they are implicitly allowed by our aggregate signature definition. Thus, other attacks that we did not study are also possible. The attack finding models and the validation model behave differently when verifying with non-honest keys. While the result in the validation models could be either true or false, the malicious aggregations of the attack finding models will, if correctly aggregated, always validate to true.

As mentioned above, the rogue public key attack violates the security definition of aggregate signatures. Thus in the default validation model, the rogue public key attack is not possible. To validate protocols that are potentially vulnerable to rogue key attacks, we create a model with additional adversary capabilities for the rogue key attack.

3.4 Summary

In this chapter, we gave an overview of our aggregate signature models. In the next two chapters, we will go into more detail and discuss the implementation in Tamarin.

The validation model can be used to verify the security properties of a protocol, allowing all subtle behaviors of aggregate signatures. As the attacks are not modeled individually, it can be difficult to reason about a found attack. Reasoning about attacks is easier with the attack finding models, as the attacks can be evaluated separately.

In our measurements, the attack finding models performed better. But all models have potential for proof time improvements. We will have a deeper look at our evaluation in Chapter 6.

Chapter 4

Attack Finding Models

In the previous chapter, we gave an overview of our models and their properties. We will now look at their implementation in Tamarin. In this chapter, we discuss the attack finding models in detail, and in the next chapter we describe the validation models. We evaluate both model classes and compare them in Chapter 6.

4.1 Naive Equation-Based Approach

To motivate our implementation, we first look at a naive approach, for which Tamarin's precomputation does not terminate.

Most of Tamarin's cryptographic primitives are modeled using functions and equations. In Section 2.2 on page 17, we provide the equational theory for signatures. The verification succeeds, if the message and secret key match the message and public key provided for the verification. We defined aggregate signatures in Definition 3.1 on page 23 in a similar manner. All the messages and public keys of the signatures in the aggregation need to match those provided for the verification. Thus, translating an aggregate signature in the same manner seems intuitive.

Analogous to Definition 3.1, we represent a signature aggregation as the function `agg/1` applied to a list of signatures. The signatures are modeled by Tamarin's built-in signature theory stated in Section 2.2. For the verification, we use the function symbol `verifyAgg/3`. The arguments are a signature aggregation, a list of messages, and a list of public keys. We formulate the verification with the following recursive equations:

```
equations: verifyAgg(agg(<sign(m1, sk),sRest>)
               , <m1, mRest>, <pk(sk), pkRest> )
         = verifyAgg(agg(sRest), mRest, pkRest)
equations: verifyAgg(agg(sign(m, sk)), m, pk(sk)) = true
```

Each application of an equation represents a verification step, where one signature is verified. The first equation describes the general case: the first signature in the aggregation is compared to the first message and the first public key. If they match, the signature is valid. We omit the signature, message and public key and the remaining signatures can be verified. The second equation represents the base case, where only one signature in the aggregation is left. If this signature is also valid, the whole aggregation was valid and we get the result `true`. Note that if at some point of the verification the signature is not valid, the equations cannot be applied and we will not reach the result `true`.

The precomputation of this model does not terminate. Informally, the problem lies in the fact, that the variables s, m, and k in the term $\texttt{vfyAgg}(\texttt{agg}(s), m, k)$ can be arbitrarily long tuples. Thus, our first verification equation can be applied arbitrarily often. Therefore, a rule containing the above term has infinitely many instantiations. When Tamarin tries to derive those instantiations in the precomputation, it will not terminate. This non-termination relates to the finite variant property, which we will not define here. See [14] Section 6 for more details.

4.2 One Verification Step

To remove the recursion in the verification, we want a single equation that is applied once. This equation should state: “The messages provided for the verification are the same as the signed messages. And the provided public keys match the secret keys used for the signatures”. We can state this by representing the aggregate signature as a list of messages and a list of public keys. The aggregation $\texttt{agg}(\texttt{sign}(m_1, sk_1), ..., \texttt{sign}(m_n, sk_n))$ is represented by

```
agg(<m1, ..., mn>, <pk(sk1), ..., pk(skn)>)
```

Note that the representation in Tamarin contains the public keys and not the secret keys. Thus, for the verification we can directly compare the public keys of the aggregate signature and the public keys provided for the verification. We define the functions `agg` and `verfiyAgg` and verification equation as follows:

```
functions: agg/2 [private]
functions: verifyAgg/3
equations: verifyAgg(agg(m_list, pk_list), m_list, pk_list) = true
```

The verification function accepts if the list of messages `m_list` and the list of public keys `pk_list` inside the aggregation function are the same as the ones provided to the verification function as the second and third arguments.

An agent should only be able to create a valid aggregation if they are in possession of the signatures. To prevent an adversary that is in possession of messages $m_1, ..., m_n$ and public keys $pk_1, ..., pk_n$ but not in possession of the corresponding signatures from creating an aggregation, the function `agg` is private. In Tamarin, private functions can only be used in rules. The adversary is not able to construct private functions. But the agent rules can directly apply the aggregate function. The following rule gives an example on how an agent can aggregate two signatures.

Example 4.1 *An agent is in possession of two signatures and aggregates them.*

```
rule Aggregator_receives_and_aggregates:
  [ A_1(sign(m1, sk1), sign(m2, sk2)) ]
-->
  [ A_2(agg(<m1, m2>, <pk(sk1), pk(sk2)>)) ]
```

The agent state fact `A_1` contains two signatures. The agent extracts the messages and secret keys and creates an aggregation with them. The aggregation is added to the new agent state fact `A_2`.

Note that this example rule cannot be applied on arbitrary terms. The pattern matching ensures that only valid signatures can be aggregated. In practice, one can aggregate arbitrary terms. Thus, the above rule contains an implicit signature verification. Some implementations and protocols might require a verification step while aggregating but others might not. This has to be kept in mind when using the aggregate signature models. Otherwise, one might miss some attacks.

In practice, anyone can aggregate signatures, also the adversary. We provide the adversary an aggregation oracle, modeled by the following two rules:

```
rule Adv_Aggregate:
  [ In(<sign(m, sk), agg(messages, keys)>) ]
-->
  [ Out(agg(<m, messages>, <pk(sk), keys>)) ]

rule Adv_Aggregate_BaseCase:
  [ In(sign(m, sk)) ]
-->
  [ Out(agg(m, pk(sk))) ]
```

The adversary accesses the oracle via `In` and `Out` facts. The aggregation is done in an incremental manner. In the first rule, the adversary provides a signature and an aggregate signature to the oracle via an `In` fact. The oracle adds the signature to the aggregation and provides the new aggregation to the adversary via an `Out` fact. The second rule represents the base case, where one signature is aggregated.

Note that this rule only allows the aggregation of valid signatures. Thus, the resulting aggregate signature is valid by construction. The same is true for example 4.1. An invalid aggregation can be modeled by an arbitrary term. The verification equation cannot be applied to `verifyAgg(term, m, k)`. Thus, the arbitrary term `term` will not validate as an aggregate signature.

In the above example 4.1, the agent aggregates two signatures in one step. This could be extended to more signatures. But it would always model the aggregation of a fixed number of signatures. We can apply the approach of the adversary aggregation oracle to the aggregation by protocol agents. The aggregation is done incrementally, in each rule one signature is added to the aggregation. This enables us to model the aggregation of arbitrarily many aggregated signatures. But it has to be kept in mind that some aggregate signatures such as BLS support incremental aggregation, while others do not. When modeling aggregate signatures that do not support incremental aggregation, one has to adapt the adversary aggregation rules.

We have so far modeled the aggregation as two tuples, one containing the messages and one containing the public keys. Alternatively, one could use a multiset of pairs where each pair contains a message and a public key. The aggregation $\mathtt{agg}(\mathtt{sign}(m_1, sk_1), \mathtt{sign}(m_2, sk_2), ..., \mathtt{sign}(m_n, sk_n))$ would be represented as

```
agg(<m1, pk(sk1)> + <m2, pk(sk2)> + ... + <mn, pk(skn)>)}
```

In a preliminary evaluation, we did not find a difference in proof time for the two approaches. We decided to use the version with two tuples as it is closer to the computational representation and thus more intuitive. Multisets in Tamarin are modeled as the associative-commutative operator +, see the Tamarin manual [22]. Thus, we can access each message and key pair by pattern matching with `agg(<mi, pki> + rest)`. In contrast, we cannot do this with Tamarin's tuples. We can only access elements with a given index; for example, we can access the third element `e3` in `<e1, <e2, <e3, rest>>>`. Depending on the protocol, one might need to access arbitrary elements, in that case, the multiset of pairs version would be better suited.

4.3 Extension for Attacks

Our model so far does not support special adversary capabilities. As described in Section 3.2, we extended our models to enable the adversary to perform colliding signature attacks or rogue public key attacks. In this section, we will describe those attack models in detail. We first introduce the public key infrastructure. And then we begin with the rogue public key attack model, as it is simpler than the colliding signature model, which we will describe afterward.

We use the key infrastructure provided by the Tamarin manual [22]:

```
rule Register_pk:
  [ Fr(~ltkA) ]
--[ RegisterHonestKey(pk(~ltkA)) ]->
  [ !Ltk($A, ~ltkA)
  , !Pk($A, pk(~ltkA))
  ,  Out(pk(~ltkA)) ]

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

The first rule models the registration of a new long-term key `~ltkA` for agent `$A`. The agent `$A` can access `~ltkA` through the fact `!Ltk`. And other agents can access the public key `pk(~ltkA)` through the fact `!Pk`. The public key is also provided to the adversary over an `Out` fact.

The second rule models a long-term key reveal. This corresponds to the agent being compromised. The adversary learns the long-term key `ltk`.

4.3.1 Rogue Public Key Attack

We described the rogue public key attack in Section 2.1.6 and defined functions to model it in Section 3.2.2. We defined a rogue public key as

$$pk_{\text{rogue}} = \texttt{roguePk}(pk_{\text{target}}) \tag{4.1}$$

and a rogue aggregation as

$$\sigma_{\text{agg}_{\text{rogue}}} = \texttt{rogueAgg}(m, pk_{\text{rogue}}) \tag{4.2}$$

We model rogue public keys with the new function `roguePk/1`. We differentiate between honest aggregations and rogue aggregations. For that, we rename the function `agg/2` to `validAgg/2` and we introduce the new private function `rogueAgg/2`. `validAgg` represents an aggregation of honest signatures and `rogueAgg` represents an aggregation that contains at least one rogue aggregation. Therefore, aggregating rogue aggregations and valid aggregations results in a rogue aggregation. The verification equation for valid aggregations stays the same and we add an equivalent verification equation for rogue aggregates.

```
equations: verifyAgg(validAgg(m, k), m, k) = true
equations: verifyAgg(rogueAgg(m, k), m, k) = true
```

As we use equivalent equations for valid and rogue aggregations, we could use one function symbol for both. But this differentiation is useful when analyzing attack traces. We recognize the rogue aggregates immediately. However, one has to be cautious when pattern matching with the aggregation functions: for example, when an agent receives an aggregate signature and we use the fact `In(validAgg(m, k))` on the left hand side of an agent rule, the agent will only receive valid aggregations. The adversary could not provide an aggregation of the form `rogueAgg(m, k)`. It would be safer to use the fact `In(aggregation)` where the variable `aggregation` could be instantiated by a `validAgg(m, k)` or `rogueAgg(m, k)`.

To enable the adversary to create rogue aggregations, we add another aggregation oracle in the form of rules:

```
rule Adv_RogueKey_Aggregation_new:
  let pkRogue = roguePk(pkTarget)
  in
  [ In(<m, pkRogue>) ]
-->
  [ Out(rogueAgg(<m, m>, <pkTarget, pkRogue>)) ]
```

To create a rogue aggregation, a message `m` and a rogue public key `roguePk(pkTarget)` are provided. Note that the adversary can create such a key for a target public key pkTarget. The oracle returns a rogue aggregation for twice the message m, the target public key pkTarget and the rogue key roguePk(pkTarget).

With this rule, the adversary can create a rogue aggregation for a target public key and a corresponding rogue public key. Such a rogue aggregate can be aggregated with other rogue aggregates or honest signatures. Therefore, we add three more oracle rules that enable the adversary to aggregate a rogue aggregation and a valid aggregation, a valid signature and a rogue aggregation, and two rogue aggregations.

The adversary can create rogue public keys for any public keys. But so far, honest agents cannot access the rogue public keys. Thus, we add the following rogue key registration rule.

```
rule Register_rogue_pk:
  [ In(roguePk(m, pkTarget)) ]
--[ Malicious($Adversary)]->
  [ !Pk($Adversary, roguePk(m, pkTarget)) ]
```

This enables honest agents to access the rogue public key over the !Pk fact.

4.3.2 Colliding Signatures

In Section 3.2.1, we defined the colliding signature attack as follows: The adversary creates malicious keys $sk_{\mathrm{mal}_1}, ...sk_{\mathrm{mal}_k}$. Signatures of those malicious keys $\mathtt{sign}(m_i, sk_{\mathrm{mal}_i})$ validate for any messages. Such malicious signatures can be aggregated with some honest signatures $\sigma_l, ..., \sigma_n$ that validate for the messages $m_l, ..., m_n$ and public keys $pk_l, ..., pk_n$. The aggregate signature of those malicious and honest signatures will validate for the messages $m_l, ..., m_n$ and any messages $m'_1, ..., m'_k$. We express this in the following equation.

$$\begin{aligned}\mathtt{vfyAgg}(\mathtt{agg}(\mathtt{sign}(m_1, sk_{\mathrm{mal}_1}), ..., \mathtt{sign}(m_k, sk_{\mathrm{mal}_k}), \sigma_l, ..., \sigma_n),\\ (m'_1, ..., m'_k, m_l, ..., m_n), (pk_{\mathrm{mal}_1}, ..., pk_{\mathrm{mal}_k}, pk_l, .., pk_n)) = \mathsf{true}\end{aligned} \tag{4.3}$$

The representation of aggregate signatures agg(m, k) that we have used so far explicitly states for which messages it will validate. But with the colliding signature attack, the messages corresponding to the malicious keys can be replaced by any messages. Consequently, we cannot use the equation:

```
verifyAgg(agg(m, k), m, k) = true
```

On first glance, the equation

```
verifyAgg(zeroAgg(m1, k), m2, k) = true
```

seems to work. But this would enable us to replace all the messages and not just those corresponding to malicious keys. Thus, we need to differentiate between the malicious and the non-malicious signatures.

We model a signature aggregation that contains a colliding signature attack with the private function `zeroAgg/2` after the splitting zero attack. The first argument is a valid aggregation and the second is a list of malicious public keys. The aggregation from the above equation 4.3 would be represented as:

```
zeroAgg(validAgg(<m1, ..., mn>, <pk(sk1), ..., pk(skn)>)
                     , <pk(skMal1), ..., pk(skMalk)>)
```

In the verification, only the messages of the valid aggregation are checked. We formulate this with the following equations.

```
equations: verifyAgg(zeroAgg(validAgg(m, k), kZero)
  , <mMalicious, m>, <kZero, k>) = true
equations: verifyAgg(zeroAgg(validAgg(m, k), <kZero1, kZero2>)
  , <mMalicious1, <mMalicious2, m>>
  , <kZero1, <kZero2, k>>) = true
```

The first equation models the case of one malicious key and the second equation represents the case of two malicious keys. The malicious and honest signatures are validated separately. The malicious keys `kZero`, `kZero1`, and `kZero2` must be the first ones in the list of keys provided for the verification and they must be the second argument of the malicious aggregation `zeroAgg`. The corresponding malicious messages `mMalicious`, `mMalicious1`, and `mMalicious2` can be arbitrary. Where the remaining messages `m` and keys `k` need to match those inside the valid aggregation `validAgg`.

With those two equations, we can only cover the case of one or two malicious keys. We can add more equations, but we cannot cover arbitrarily many malicious signatures. However, a fixed number of malicious signatures is sufficient for an attack finding model. With those two equations, Tamarin can find the splitting zero attack described by Quan [19].

To create the private function `zeroAgg` and to register malicious keys, we add rules similar to those in the rogue public key model. In both the rogue public key attack model and the colliding signature model, only the adversary can aggregate malicious signatures. An honest agent cannot accidentally aggregate some malicious signatures. In our weather protocol example this does not matter as the aggregator only performs the aggregation and this can also be done by the adversary. But for other protocols, where the aggregator performs some additional operations, for example encrypting the aggregate signature, one would need to add additional rules for an aggregator that accidentally aggregates malicious signatures.

Chapter 5

Validation Models

In Section 3.3, we demonstrated the need for validation models and described them broadly. In this chapter, we describe our implementation in detail. In the next chapter, we evaluate our models, compare it to the attack finding models, and evaluate the performance of our models.

In the attack finding models we presented so far, the verification and adversary capabilities are modeled through equations and rules. Each additional adversary capability has to be added explicitly. We now present an opposite approach that follows the computational definition of aggregate signatures. The correctness and security definition describe the verification results for honestly generated keys whereas the verification result for non-honest keys is only restricted through the deterministic nature of the verification algorithm. Thus, every attack that is not explicitly forbidden by the cryptographic definition is possible. We model this using Tamarin's restrictions. Jackson et al. [17] first introduced this method of using Tamarin's restrictions to model cryptographic primitives.

Restrictions restrict which traces Tamarin analyzes. Let us look, as an example, at the following restriction, which is adapted from the Tamarin manual [22]:

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

This restriction ensures that Tamarin only considers traces, where in all appearances of the fact `Equal`, the first and second arguments are equal. We can use this, for example, to model a signature verification. One adds the action fact

```
Equal(verify(signature, m, pk), true)
```

to the agent rule where the verification takes place. Thereby, Tamarin only considers traces where the verification `verify(signature, m, pk)` is equal to `true`.

We extend this idea by translating the computational definition of aggregate signatures into Tamarin restrictions. We formulate the restrictions around the action fact

`VerifyAgg`. This fact contains the signature aggregation, the messages and public keys, and the expected verification result. In most cases, the expected verification result is true, but if a protocol expects a certain behavior in case of an invalid signature, one can use the new nullary function `false/0`.

5.1 Formalizing the Restrictions

We now formulate the correctness, security, and consistency of aggregate signatures, motivated in Section 3.3, more formally.

5.1.1 Correctness

The correctness definition of aggregate signatures 2.4 on page 8 states that if all signatures in an aggregation are accepted, the aggregation has to be accepted as well:

$$\forall \boldsymbol{pk}, \boldsymbol{m}, \boldsymbol{\sigma}.(\forall i \in \{1, ..., n\}.\mathtt{vfy}(\sigma_i, m_i, pk_i) = \mathsf{true}) \\ \implies \mathsf{vfyAgg}(\mathsf{agg}(\boldsymbol{\sigma}), \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true} \quad (5.1)$$

where $\boldsymbol{pk} = (pk_1, ..., pk_n)$, $\boldsymbol{m} = (m_1, ..., m_n)$ and $\boldsymbol{\sigma} = (\sigma_1, ..., \sigma_n)$

We want to describe the result of vfyAgg independently of the result of vfy. For that we use the correctness definiton of signatures, which states: For a key pair (sk, pk) output by the key generation algorithm and all messages m, we have:

$$\mathtt{vfy}(\mathtt{sign}(m, sk), m, pk) = \mathsf{true} \quad (5.2)$$

Analogous to Jackson et al. [17], we use the predicate $\text{Honest}(pk)$ to state that the public key pk was honestly generated by the key generation algorithm.

As $\text{Honest}(pk) \wedge \sigma = \mathtt{sign}(sk, m)$ implies that the signature verification succeeds, we can formulate the following correctness restriction.

Restriction 5.1 (Correctness of aggregate signatures)

$$\forall \boldsymbol{pk}, \boldsymbol{m}, \boldsymbol{\sigma}.(\forall i \in \{1, ..., n\}.(\text{Honest}(pk_i) \wedge \sigma_i = \mathtt{sign}(m_i, sk_i)) \\ \implies \mathsf{vfyAgg}(\mathsf{agg}(\boldsymbol{\sigma}), \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true} \quad (5.3)$$

5.1.2 Security

We will first derive our security restriction from the computational security definition and then explain the restriction on an example.

The existential unforgeability definition of aggregate signatures (Definition 2.6 on page 8) states that no adversary, capable of winning the attack game with an non-negligible probability, exists. We abstract this and assume that no adversary can win the attack game. Winning the attack game means, after having access to a signing oracle, the adversary produces a forgery $(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{agg}})$ where $\texttt{vfyAgg}(\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}) = \texttt{true}$ where one of the public keys $pk_j \in \boldsymbol{pk}$ is the honestly generated one provided by the challenger, the corresponding secret key sk_j is not known to the adversary, and the corresponding message $m_j \in \boldsymbol{m}$ was not queried by the adversary. As we assume that such a forgery is not possible, for each triple $(\boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{agg}})$ where $\texttt{vfyAgg}(\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}) = \texttt{true}$, each public key $pk_i \in \boldsymbol{pk}$ is either not honest or there is an corresponding signature σ_i that is honestly created, which means $\sigma_i = \texttt{sign}(m_i, sk_i)$. Note that σ_{agg} can be an arbitrary term if all public keys are not honest. Only if some public keys are honest, it has to be an aggregation of signatures $\sigma_{\text{agg}} = \texttt{agg}(\boldsymbol{\sigma})$. We express this in the following equation, which we will provide an example for:

$$\begin{aligned}
&\forall \boldsymbol{pk}, \boldsymbol{m}, \sigma_{\text{agg}}.\texttt{vfyAgg}(\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}) = \texttt{true} \\
&\qquad\qquad\qquad \implies (\forall i.\neg\text{Honest}(pk_i) \vee (\exists \boldsymbol{\sigma}.\sigma_{\text{agg}} = \texttt{agg}(\boldsymbol{\sigma}) \\
&\qquad\qquad\qquad\quad \wedge (\forall i.\neg\text{Honest}(pk_i) \vee \exists \sigma_i \in \boldsymbol{\sigma}.\sigma_i = \texttt{sign}(m_i, sk_i)))) \quad (5.4)
\end{aligned}$$

As stated above, if no public key is honest ($\forall i.\neg\text{Honest}(pk_i)$), there are no security guarantees. Otherwise, there has to be a list of signatures $\boldsymbol{\sigma}$, such that $\sigma_{\text{agg}} = \texttt{agg}(\boldsymbol{\sigma})$. We now explain the last part of the restriction on the following example:

Example 5.2 *We verify an aggregate signature* $\texttt{agg}(\sigma_1, \sigma_2)$ *on two messages* m_1, m_2 *and two public keys, one honest public key* pk_{honest} *and a maliciously created public key* $pk_{malicious}$*. The verification returns* $\texttt{true}$*:*

$$\texttt{vfyAgg}(\texttt{agg}(\sigma_1, \sigma_2), (m_1, m_2), (pk_{honest}, pk_{malicious})) = \texttt{true} \quad (5.5)$$

The restriction now states that each public key has to be non-honest or the corresponding signature has to be valid. We first look at our honest public key pk_{honest}*. As it is honest,* σ_1 *has to be* $\texttt{sign}(m_1, sk_{honest})$ *where* sk_{honest} *is the secret key of* pk_{honest}*. Now for the malicious key* $pk_{malicious}$*, the corresponding signature* σ_2 *can be anything as the public key is not honest.*

To simplify the security restriction, we make the following assumptions:

- The message and public key vectors $\boldsymbol{m}$ and $\boldsymbol{pk}$ need to be of the same length.
- The signature aggregation σ_{agg} needs to be of the form $\texttt{agg}(\boldsymbol{\sigma})$.
- There must be a signature σ_i for each message and key pair (m_i, pk_i) and vice versa.

We justify those assumptions with the fact that, in practice, a verification algorithm first does some input validations. For example, the IETF draft for BLS aggregate signatures [9] requires the public key and message vector to be of the same length and stipulates multiple additional checks such as checking the subgroup of the signature. We enforce the above assumptions with some additional restrictions, which we will discuss further in Section 6.2.2. With those assumptions, we can simplify Equation 5.4 and get the following security restriction:

Restriction 5.3 (Security of aggregate signatures)

$$\begin{aligned}\forall \boldsymbol{pk}, \boldsymbol{m}, \boldsymbol{\sigma}.\mathtt{vfyAgg}(\mathtt{agg}(\boldsymbol{\sigma}), \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{true} \\ \implies (\forall j.(\neg \text{Honest}(pk_j) \vee \sigma_j = \mathtt{sign}(m_j, sk_j)))\end{aligned} \tag{5.6}$$

We add additional restrictions to enforce our above-mentioned three assumptions.

5.1.3 Consistency

We stated in Section 3.3 that the verification with non-honest keys can either be true or false. But due to the determinism of the verification algorithm, the same verification needs to always have the same result. We express this in the following consistency restriction:

Restriction 5.4 (Consistency of aggregate signatures)

$$\begin{aligned}\forall \boldsymbol{pk}, \boldsymbol{m}, \sigma_{agg}, b_1, b_2.\mathtt{vfyAgg}(\sigma_{agg}, \boldsymbol{m}, \boldsymbol{pk}) = b_1 \\ \wedge \mathtt{vfyAgg}(\sigma_{agg}, \boldsymbol{m}, \boldsymbol{pk}) = b_2 \Rightarrow b_1 = b_2\end{aligned} \tag{5.7}$$

5.2 Signature Aggregation in Tamarin

In this section, we discuss how we represent the signature aggregation and verification in detail.

The above-mentioned correctness and security restrictions all quantify over the signatures in the aggregation. Thus, we need to be able to express this quantification. So far, we have used tuples to model the lists of signatures, messages, and public keys. As mentioned in Section 4.2, we cannot access elements of an arbitrary index in tuples. This is, however, possible with multisets. The use of multisets makes it for the restrictions possible to quantify over the signatures, messages, and keys. We show how we express those quantifications in the next section.

A drawback of multisets compared to tuples is that the elements are not ordered due to their associativity-commutativity. Our restrictions contain statements on the public keys, messages, and signatures of a certain index. We need a binding between σ_i, m_i, and pk_i. Thus, we add an explicit index. We add this index to each signature, message, and public key.

The aggregation is represented by the function `agg/1` applied to a multiset of tuples of signatures and indexes. For example, we model

$$\mathtt{agg}(\mathtt{sign}(m_1, pk_1), \mathtt{sign}(m_2, pk_2)) \tag{5.8}$$

as

```
agg(<sign(m1, pk1), ind_1> + <sign(m2, pk2), ind_2>)
```

The public keys and messages provided for the verification are represented as a multiset of tuples, each being a triple of a message, a public key, and an index. Let us look at the following example:

Example 5.5 *When validating some signature aggregation* `agg` *on some messages* `m1, m2` *and public keys* `pk1, pk2`*, expecting the result* true *one adds the following action fact to the corresponding agent rule.*

```
  VerifyAgg(agg, <m1, pk1, index_1> + <m2, pk2, index_2>, true)
```

We now discuss how to model the index. The index of an honest aggregation needs to have the following two properties:

1. The adversary needs to be able to provide and create the index as the adversary needs to be able to aggregate signatures and provide messages and keys for the verification.
2. Each index needs to be distinct. There cannot be multiple message and key pairs for one signature or vice versa.

We considered the following options to represent the index:

- fresh value
- constants, such as `'1', '2', '3', ...`
- counter, such as `'1', '1'+'1', '1'+'1'+'1', ...`
- a value provided by the adversary
- the public key and messages

We did not find a definite best representation for the index. There are different advantages and disadvantages for each option. We focused on the representation as a constant and a fresh value. A constant is the most intuitive representation but it has the disadvantage that it has to be stated explicitly. We cannot use it to model arbitrary large aggregations. On the other hand, we can model arbitrary large aggregations with fresh values.

The adversary can construct each constant. When using fresh values, one has to provide the fresh value to the adversary after the aggregation. To ensure that the indexes in one aggregation are distinct, we add restrictions.

The function `agg/1` is public. Thus, anyone, honest agents and the adversary, can aggregate the signatures in their possession. In addition, we can aggregate any term, valid signature or arbitrary symbol. This makes this model a lot more flexible than the attack models described in Chapter 4. But the aggregation has to be done in one step. Incremental aggregation is not supported. However, the model can be extended.

5.3 Restrictions in Tamarin

In the above two sections, we defined our restrictions and discussed how we represent signature aggregations in Tamarin. We now apply those concepts to formulate the restrictions in Tamarin notation. We discuss the correctness restriction as an example. We reformulate Restriction 5.1 on page 38 to the following equivalent restriction:

Restriction 5.6

$$\begin{aligned}\forall \boldsymbol{pk}, \boldsymbol{m}, \boldsymbol{\sigma}.\texttt{vfyAgg}(\texttt{agg}(\boldsymbol{\sigma}), \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{false} \\ \implies (\exists i \in \{1, ..., n\}.(\neg\sigma_i = \texttt{sign}(m_i, sk_i) \vee \neg\text{Honest}(pk_i)))\end{aligned} \tag{5.9}$$

We translate this into the following Tamarin restriction which we will discuss in detail:

```
1  restriction Verification_Correctness_morePrecise:
2    "All aggregation mAndPk #i.
3     VerifyAgg(aggregation, mAndPk, false)@i
4     ==>
5      ((Ex si ind thetaAgg mi ski thetaMPk.
6          VerifyAgg(agg(<si, ind>+thetaAgg)
7            , <mi, pk(ski), ind> + thetaMPk, false)@i
8        & (not(si = sign(mi, ski))
9          | not(Ex #j. RegisterHonestKey(pk(ski))@j)))
10     |(Ex si ind mi ski.
11         VerifyAgg(agg(<si, ind>), <mi, pk(ski), ind>, false)@i
12       & (not (si = sign(mi, ski))
13         | not(Ex #j. RegisterHonestKey(pk(ski))@j))))"
```

Lines 2 to 3 are the left hand side of the implication. We express

$$\texttt{vfyAgg}(\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}) = \mathsf{false} \tag{5.10}$$

with the action fact `VerifyAgg(aggregation, mAndPk, false)`. We quantify over the occurrences of verification with the result false and over all signature aggregations, messages, and public keys. In other words, for each trace that contains an action fact `VerifyAgg` with the verification result false, the right hand side expressed in lines 5 to 13 must hold.

For the right hand side, we have to do a case distinction. Lines 5 to 9 treat the case of two or more aggregated signatures and lines 10 to 13 the case of one aggregated signature. To explain why we need this distinction, we first have a look at how we express the existential quantification of the right hand side. The existential quantification $\exists i \in \{1, ..., n\}$ quantifies over the signatures, messages and secret keys: $\exists \sigma_i \in \boldsymbol{\sigma}, m_i \in \boldsymbol{m}, sk_i \in \boldsymbol{sk}$. We now look at how we express $\exists \sigma_i \in \boldsymbol{\sigma}$. The aggregation $\texttt{agg}(\boldsymbol{\sigma})$ is represented as

```
agg(<s1, ind1>+ ... +<sn, indn>)
```

Due to the associative-commutative property of multisets, this matches

```
agg(<si, ind>+thetaAgg)
```

See the following equation:

$$\begin{aligned} &\langle \sigma_1, ind_1 \rangle + \langle \sigma_2, ind_2 \rangle + ... + \langle \sigma_n, ind_n \rangle \\ \equiv_{AC}\ &\langle \sigma_i, ind_i \rangle + \underbrace{\langle \sigma_1, ind_1 \rangle + ... + \langle \sigma_{i-1}, ind_{i-1} \rangle + \langle \sigma_{i+1}, ind_{i+1} \rangle + ... + \langle \sigma_n, ind_n \rangle}_{\vartheta_{agg}} \end{aligned} \tag{5.11}$$

Thus we can express the existential quantification $\exists \sigma_i \in \boldsymbol{\sigma}$ with

```
Ex si ind thetaAgg. aggregation = agg(<si, ind>+thetaAgg)
```

where `thetaAgg` is a variable that can be instantiated by the remaining signatures. Instead of using this equality, we restate the action fact `VerifyAgg` in lines 5 and 6 with `aggregation` replaced with `agg(<si, ind>+thetaAgg)`. We discuss those different formulations at the end of this section.

Now back to the case distinction: In our model, we do not use an empty element in the multisets. `thetaAgg` will be instantiated by a multiset or a tuple `<si, ind>`. Thus, with `agg(<si, ind>+thetaAgg)`, we can only represent signature aggregations with two or more elements. Therefore, we need to cover the case of one aggregated signature separately. See line 11, where the aggregation of one signature is represented by `agg(<si, ind>)`. In other words, we need the case distinction, as `agg(<si, ind>+thetaAgg)` and `agg(<si, ind>)` do not pattern match. For other restrictions, we had to add a second restriction to cover the case of one aggregated signature.

The rest of the implication $\neg \sigma_i = \texttt{sign}(m_i, sk_i) \vee \neg \text{Honest}(pk_i)$ can be translated straightforwardly in lines 8, 9, 12, and 13.

We noted above, that we can express $\sigma_i \in \boldsymbol{\sigma}$ by stating

```
aggregation = agg(<si, ind>+thetaAgg)
```

or by restating the action fact where `aggregation` is replaced by `agg(<si, ind>+thetaAgg)`. Let us look, as an example, at the following two restrictions:

```
restriction OneMessageKeyPairPerSignature_RestateActionFact:
  "All si thetaAgg ind messagesKeys #i.
    VerifyAgg(agg(<si, ind>+thetaAgg), messagesKeys, true)@i
  ==> Ex mi ski thetaMPk.
    VerifyAgg(agg(<si, ind>+thetaAgg)
      , <mi, pk(ski), ind> + thetaMPk, true)@i"

restriction OneMessageKeyPairPerSignature_Equation:
  "All si thetaAgg ind messagesKeys #i.
    VerifyAgg(agg(<si, ind>+thetaAgg), messagesKeys, true)@i
  ==> Ex mi ski thetaMPk.
    messagesKeys = <mi, pk(ski), ind> + thetaMPk"
```

The two restrictions are equivalent. They ensure that there is a signature for each message and key pair. Note that the first one restates the action fact `VerifyAgg` after the implication, while the second explicitly equates `messagesKeys` and `<mi, pk(ski), ind>+theatMPk`. The two formulations are equivalent. But interestingly, using the first formulation results in non-termination. Exploring how different formulations of the same restriction are treated by Tamarin would be interesting for future work.

5.4 Extension for Attacks

As the correctness and security restriction only cover verifications with honest keys, we can enable various attacks, by adding the ability to register malicious keys. Similar to the malicious and rogue keys described in Section 4.3, we add the following rules for the adversary to register malicious keys:

```
rule Register_malicious_pk:
  [ Fr(~ltkMalicious) ]
--[ Malicious($A) ]->
  [ !Ltk_malicious($A, ~ltkMalicious)
  , !Pk($A, pk(~ltkMalicious))
  , Out(pk(~ltkMalicious)) ]

rule Reveal_malicious_ltk:
  [ !Ltk_malicious(A, ltkMalicious)]
--[ LtkReveal(A) ]->
  [ Out(ltkMalicious) ]
```

In contrast to the honest key registration rule, stated in Section 4.3, we here create a malicious key, instead of an honest key. To prevent that honest agents use the malicious keys as their own, we call the long-term key fact `Ltk_malicious`. The adversary gets the malicious keys through the reveal rule. And any honest agent can access the malicious public keys through the `!Pk` fact.

The addition of those two rules is sufficient to model the splitting zero attack. We will discuss the possible attacks in more detail in Section 6.1

5.4.1 Restriction-Based Rogue Public Key Model

The rogue key attack violates the aggregate signature security definition. Thus our validation model does not allow a rogue key attack. To enable the attack, we add an attack oracle, similar to the one presented for the attack finding model in Section 4.3.1. In contrast to the attack finding model, the aggregation function contains the signatures and not only the messages and public keys. So when creating an attack oracle, we need access to the rogue secret key. Note that in practice the attacker creates the rogue public key without the secret key. So in our model, the attacker cannot learn the rogue secret key. We replace the public function `roguePk/1` with the private function `rogueSk/1`. We then add the following rule to register a rogue key:

```
rule Register_RogueKey:
  let pkRogue = pk(rogueSk(pkTarget))
  in
  [ In(pkTarget) ]
--[ Register($Adversary, pkRogue)
  , Malicious($Adversary) ]->
  [ !Pk($Adversary, pkRogue)
  , Out(pkRogue) ]
```

The adversary requests the rogue public key for a target public key `pkTarget`. The registration rule provides the rogue public key `pk(rogueSk(pkTarget))` without revealing the rogue secret key.

The adversary can create rogue public key aggregations with an attack oracle, similar to the one for the attack finding model, described in Section 4.3.1. The adversary provides a rogue secret key `pk(rogueSk(pk(skTarget)))` and a message `m`. The attack oracle returns the corresponding rogue public key aggregation:

```
agg(<sign(m, skTarget), index_target>
  + <sign(m, rogueSk(pk(skTarget))), index_rogue>)
```

Note that the oracle extracts the target secret key `skTarget` and the rogue secret key `rogueSk(pk(skTarget))` to create the rogue aggregation, while the adversary has access to neither of the two secret keys. We provide an additional attack oracle rule to aggregate additional signatures to the rogue aggregate. Note that the rogue public key is not an honest key and thus it can be used by the adversary to perform all attacks that are possible with a regular malicious key.

Chapter 6

Evaluation

In this chapter, we discuss the properties of our aggregate signature models. We structure this discussion in three sections:

In the first section, we look at what influence the different aggregate signature models have on the properties of our example weather protocol. In Section 6.2, we contrast the behavior of our two approaches, the attack finding and validation models, by comparing the verification results for different aggregations. And in the last section of this chapter, we evaluate the proof times and attack finding times of different lemmas on our example weather protocol.

6.1 Comparing the Models

In this section, we summarize the properties of our attack finding and validation models. We first describe four lemmas which we then use to point out the differences between our models.

6.1.1 Lemmas

To evaluate our models, we use the example weather protocol described in Section 3.1. We describe the protocol properties by formulating four lemmas: two authentication lemmas, which we described in Section 3.1.1 and which we will now phrase more formally, and two new lemmas that each rule out one of our attacks.

To formulate our lemmas, we need to add action facts to our protocol. Figure 6.1 shows the message sequence chart of the protocol with the additional action facts Running, Verify, Commit, and ClaimHonest.

We add the action fact $\mathrm{Running}(S_i, V, m_i)$ to the station roles. The four action facts, $\mathrm{Verify}(\sigma_{\mathrm{agg}}, \boldsymbol{m}, \boldsymbol{pk})$, $\mathrm{Commit}(V, S_i, m_i)$, $\mathrm{ClaimHonest}(V)$, and $\mathrm{ClaimHonest}(S_i)$, are stated one after another on the verifier role; however, the four action facts happen simultaneously.

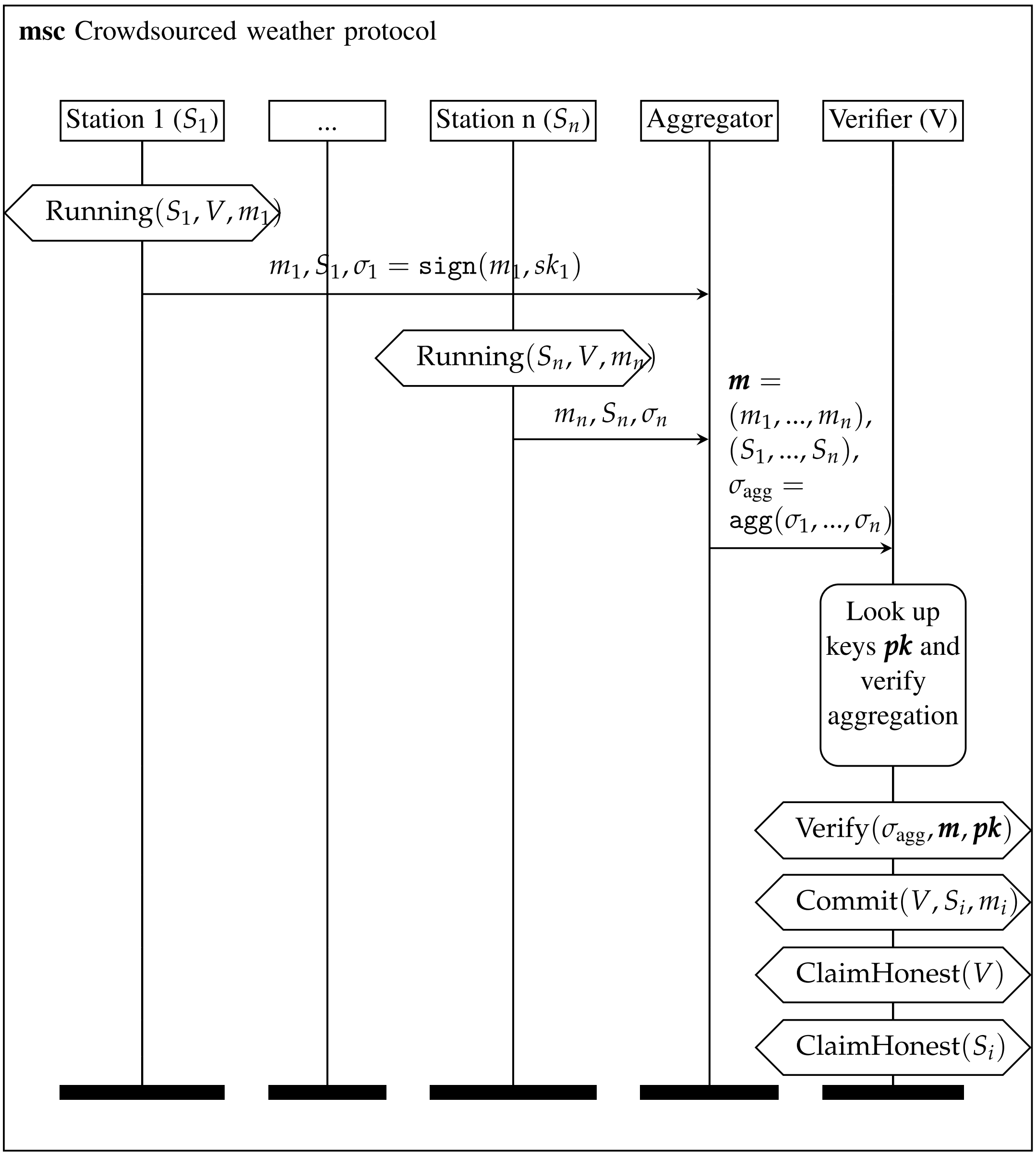
msc Crowdsourced weather protocol
Station 1 (S_1)
...
Station n (S_n)
Aggregator
Verifier (V)
Running(S_1, V, m_1)
$m_1, S_1, \sigma_1 = \texttt{sign}(m_1, sk_1)$
Running(S_n, V, m_n)
m_n, S_n, σ_n
$\boldsymbol{m} = (m_1, ..., m_n), (S_1, ..., S_n), \sigma_{\text{agg}} = \texttt{agg}(\sigma_1, ..., \sigma_n)$
Look up keys $\boldsymbol{pk}$ and verify aggregation
Verify($\sigma_{\text{agg}}, \boldsymbol{m}, \boldsymbol{pk}$)
Commit(V, S_i, m_i)
ClaimHonest(V)
ClaimHonest(S_i)

Figure 6.1: Message sequence chart of the synthetic crowdsourced weather data protocol with action facts

We now formulate the two authentication lemmas from Section 3.1.1 using those action facts.

Message authenticity states that for each message m, signed with a key of sensor S_i and received by a verifier V, the sensor S_i must have signed the message m or one of the agents S_i or V is not honest.

Lemma 6.1 (Message authenticity)

$$\begin{aligned} &\forall V_i, S, m, i.\text{Commit}(V_i, S, m)@i \\ &\qquad\qquad \implies (\exists V_j, j.\text{Running}(S, V_j, m)@j) \\ &\qquad\qquad \vee (\exists C, r.\text{LtkReveal}(C)@r \wedge \text{ClaimHonest}(C)@i) \\ &\qquad\qquad\qquad \vee (\exists C, r, \text{Malicious}(C)@r \wedge \text{ClaimHonest}(C)@i) \end{aligned} \tag{6.1}$$

The fact LtkReveal relates to a compromised agent as shown in the rule `Reveal_ltk` in Section 4.3 on page 32. And the action fact Malicious refers to the registration of malicious keys, such as the registration of rogue keys in Section 4.3.1.

Note that the Commit and Running facts contain two different verifier names V_i and V_j. Thus there is no statement on which destination the sensor S intends. We include the intended destination in the following standard weak agreement property:

Lemma 6.2 (Weak agreement)

$$\begin{aligned} &\forall V, S, m_i, i.\text{Commit}(V, S, m_i)@i \\ &\qquad\qquad \implies (\exists j.\text{Running}(S, V, m_j)@j) \\ &\qquad\qquad \vee (\exists C, r.\text{LtkReveal}(C)@r \wedge \text{ClaimHonest}(C)@i) \\ &\qquad\qquad\qquad \vee (\exists C, r, \text{Malicious}(C)@r \wedge \text{ClaimHonest}(C)@i) \end{aligned} \tag{6.2}$$

Note that this lemma contains no statement on the received message m_i and the sent message m_j. As discussed in Section 3.1.1, the weak agreement property does not hold for our protocol as the stations do not include the intended destination in their message.

We formulate two additional lemmas for the attacks covered by the extended models. The first lemma states the absence of a splitting zero attack. With the splitting zero attack, an aggregate signature σ_{agg} validates for different messages. Thus, our lemma states that there cannot be action facts Verify with the same aggregate signature σ_{agg} but with different messages $\boldsymbol{m}_a$ and $\boldsymbol{m}_b$.

Lemma 6.3 (No splitting zero attack)

$$\begin{aligned} &\neg(\exists \sigma_{agg}, \boldsymbol{m}_a, \boldsymbol{m}_b, \boldsymbol{k}, i, j.\text{Verify}(\sigma_{agg}, \boldsymbol{m}_a, \boldsymbol{k})@i \\ &\qquad\qquad\qquad \wedge \text{Verify}(\sigma_{agg}, \boldsymbol{m}_b, \boldsymbol{k})@j \wedge \boldsymbol{m}_a \neq \boldsymbol{m}_b) \end{aligned} \tag{6.3}$$

The last lemma states that no rogue public key attack is possible. The rogue public key attack enables an adversary to forge a signature. The verifier could receive a forged aggregate signature that validates for a message m and the public key of a sensor S. But the sensor S has not signed this message m and the sensor S was not compromised. The following lemma expresses that this described attack cannot occur.

Lemma 6.4 (No rogue key attack)

$$\neg(\exists V_i, S, m, i.\text{Commit}(V_i, S, m)@i \land \neg(\exists V_j, j.\text{Running}(S, V_j, m)@j) \\ \land \neg(\exists k.\text{LtkReveal}(S)@k) \land \neg(\exists l.\text{Malicious}(S)@l)) \quad (6.4)$$

6.1.2 Results

In Chapter 3, we gave an overview of our models and summarized how the weather protocol behaves assuming the different aggregate signature models. We now present the concrete results of letting Tamarin prove or disprove the four lemmas for each model[1]. We compare the following three attack finding models:

1. without additional adversary capabilities (See Section 4.2)
2. allowing colliding signatures (See Section 4.3.2)
3. allowing the rogue key attack (See Section 4.3.1)

And the following validation models:

4. without malicious keys (See Sections 5.1 to 5.3)
5. with rules to register malicious keys (See Section 5.4)
6. allowing the rogue key attack (See Section 5.4.1)

Table 6.1 summarizes the results:

As described in Section 3.1.1, the weak agreement property cannot hold for our weather protocol as the stations do not state the intended destination. Thus the weak agreement property does not hold for any of the models. The other three lemmas are proven for the models without additional adversary capabilities, for the attack finding model 1 and the validation model 4. The *aliveness* lemma holds as expected and the two attacks, splitting zero and rogue public key, are not possible.

The attack finding model 2 with colliding signature attack and the validation model 5 with malicious keys also behave the same on our lemmas. They both additionally allow a colliding signature attack, which results in the falsification of the *no splitting zero* lemma. What the table does not show, is that the validation model allows various additional attacks,which we have not as yet addressed. In Section 6.2.2 for example,

[1] As described in Section 6.3, not all proofs and attack searches terminated for all models and lemmas. Thus, we performed the proofs with restrictions that limit the number of aggregated signatures to two. We discuss and justify this approach in Section 6.3.

		Lemma			
Model		Message authenticity	Weak agreement	No splitting zero attack	No rogue key attack
1	Attack finding model	Proven	Falsified	Proven	Proven
2	Attack finding model colliding signatures	Proven	Falsified	Falsified	Proven
3	Attack finding model rogue public key	Falsified	Falsified	Proven	Falsified
4	Validation model	Proven	Falsified	Proven	Proven
5	Validation model with malicious keys	Proven	Falsified	Falsified	Proven
6	Validation model rogue public key	Falsified	Falsified	Falsified	Falsified

Table 6.1: Proven and falsified lemmas for all models

we will introduce an attack on the DEO property, which is possible with the validation model 5 but not with the attack finding model 2. In other words, we can formulate additional lemmas that falsify for the validation model 5 and are proven for the attack finding model 2.

In contrast to models 1, 2, 4, and 5, the rogue public key models 3 and 6 do not behave the same regarding our four lemmas. They both enable the rogue public key attack and thus lemma *no rogue key attack* is falsified. As this is an attack on the *message authenticity* lemma, it is also falsified. The difference between the two models 3 and 6 is pointed out by lemma *no splitting zero attack*. The attack finding model 3 explicitly allows the rogue public key attack. A rogue aggregation will only verify for one specific message and not, as required for the splitting zero attack, for multiple messages. Thus, the splitting zero attack is not possible for the rogue public key attack finding model 3. On the other hand, there is an attack on the *no splitting zero attack* lemma for the validation model 6 with rogue public keys. However, the validation model 6 does not have the malicious key rules of model 5. Furthermore, the rogue public keys are not honest keys and thus they can serve as malicious keys for the splitting zero attack, which leads to the falsification of lemma *no splitting zero attack*.

6.2 Correctness Evaluation

As stated in Section 5.3, it was necessary to cover the case of one signature in the restrictions for the validation models separately. This and other details can easily be overlooked. We found them due to our correctness evaluation, which we performed repetitively during the development of our models.

We evaluate the models on individual aggregate signature verifications. In the next section, we discuss how we construct the model for this correctness evaluation and in Section 6.2.2, we discuss the results.

6.2.1 Setup

We represent each aggregate signature verification with a rule. Here is an example for both the attack finding and validation model:

Example 6.5 *We evaluate the verification result for a single aggregated signature that is validated with the correct message and key pair:*

$$\mathtt{vfyAgg}(\mathtt{agg}(\mathtt{sign}(m, sk)), m, pk(sk)) \tag{6.5}$$

For this verification and the expected result true*, we formulate a multiset rewriting rule. The rules differ for the validation and attack finding model.*

For the validation model, we use the verification fact VerifyAgg*, on which we formulated the restrictions, stated in Section 5.1. Here we use the constant* '1' *as index:*

```
rule OneValidSignature_ValidationModel:
  [ Fr(~m), !Ltk($A, ~sk) ]
--[ VerifyAgg(agg(<sign(~m, ~sk), '1'>), <~m, pk(~sk), '1'>, true),
  OneValidSignature() ]->
  [ ]
```

For the attack finding model, we use the action fact Equal*. By adding the equality restriction, stated in Chapter 5 on page 37, we can ensure that this fact only occurs in a trace, if the verification result is equal to* true*. The verification is modeled with the function* verifyAgg *which is evaluated with the equation stated in Section 4.2 on page 30.*

```
rule OneValidSignature_AttackFindingModel:
  [ Fr(~m), !Ltk($A, ~sk) ]
--[ Equal(verifyAgg(validAgg(~m, pk(~sk)), ~m, pk(~sk)), true),
  OneValidSignature() ]->
  [ ]
```

Note that both rules have an additional action fact OneValidSignature. Each rule contains such a distinct action fact on which we formulate a lemma. Depending on the verification, the lemma states whether this action fact can occur or not. Let us look at the lemma for the above rules:

Example 6.6 *The following lemma states that the above evaluation can occur in a trace:*

```
lemma OneValidSignature:
 exists-trace
  "Ex #i. OneValidSignature()@i"
```

The lemma states that there must be a trace that contains the action fact OneValidSignature*. Such a trace would also contain the* VerfyAgg *or* Equal *fact from above. This means that we expect that the aggregate signature* $\mathtt{agg}(\mathtt{sign}(m, sk))$ *validates for message* m *and secret key* sk *to* true*.*

We also evaluate each verification with the result false. For that we duplicate each rule, state the verification result false, and formulate an additional lemma as follows:

Example 6.7 *For the validation model, we duplicate the rule from example 6.5 but with the verification result* false *instead of* true.

```
rule OneValidSignature_ValidationModel:
  [ Fr(~m), !Ltk($A, ~sk) ]
--[ VerifyAgg(agg(<sign(~m, ~sk), '1'>), <~m, pk(~sk), '1'>, false),
  OneValidSignature_false() ]->
  [ ]
```

For the attack finding models we need a different approach as there is no explicit evaluation to false*. A verification is* false *if there is no evaluation to* true*. Thus, we replace the* `Equal` *fact with* `NotEqual`*:*

```
rule OneValidSignature_false:
  [ Fr(~m), !Ltk($A, ~sk) ]
--[ NotEqual(verifyAgg(validAgg(~m, pk(~sk)), ~m, pk(~sk)), true),
  OneValidSignature_false() ]->
  [ ]
```

And add the following restriction:

```
restriction NoEquality:
"All v b #i. NotEqual(v, b)@i ==> not (v = b)"
```

For the attack finding model, the function `verifyAgg` *will evaluate to* true*. And for the validation model, the correctness restriction states, that in case of the verification result* false*, there must be a non-honest public key or an incorrect signature. As this is not the case, this verification with result* true *cannot occur in any trace. Thus we formulate the following lemma, which states, that the action fact of the above rules cannot occur in any trace:*

```
lemma OneValidSignature_false:
  "not (Ex #i. OneValidSignature_false()@i)"
```

We differentiate between two main groups of verifications:

Valid verifications expect to verify
e.g.: $\texttt{vfyAgg}(\texttt{agg}(\texttt{sign}(m, sk)), m, pk(sk))$

Invalid verifications expect to not verify
e.g.: $\texttt{vfyAgg}(\texttt{agg}(\texttt{sign}(m1, sk)), m2, pk(sk))$ where $m1 \neq m2$

For each verification and verification result, we formulate a lemma that either states that this verification exists, such as in Example 6.6, or a lemma that states that such a verification cannot occur, such as in the lemma in Example 6.7. Table 6.2 shows which lemma is formulated for which group of verifications and which verification result.

We provide a script that generates the theory from the above-described rules and the aggregate signature model, runs the proofs, and highlights the results.

	Verification result = true	Verification result = false
Valid verification	Trace exists	No occurrence
Invalid verification	No occurrence	Trace exists

Table 6.2: The appropriate lemma to formulate for which group of verifications and verification result

	Result = true	Result = false
Not honest keys	Trace exists	Trace exists
Wrong formats	No occurrence	No occurrence
Missing numbers	Trace exists	No occurrence

Table 6.3: Special groups of verifications and the corresponding lemmas for the verification results

6.2.2 Results

The correctness evaluation proved to be a useful tool to evaluate the behavior of our aggregate signature models. This was especially true for the validation models where we could discover edge cases that had to be covered separately.

Our correctness evaluation also reveals the differences between the attack finding and validation models.

For the attack finding models, we have the two groups of verifications that we mentioned above: valid signatures and invalid signatures. The result of a verification is either true or false.

The validation models behave a bit differently. Without the restrictions, every verification of any aggregation with any messages and keys and any verification result can appear in a trace. And thus, every verification could result in true or false. By adding restrictions, we exclude some of those traces: namely, only valid aggregations with matching messages and keys can occur in a verification fact with true and only invalid aggregations can occur in a verification fact with false.

One would expect that an aggregation with some messages and keys would either only occur in valid verifications or in invalid ones. There are, however, some special cases where this is not the case. We summarize them in Table 6.3 and describe them in detail in the following sections:

Non-Honest Keys

As stated in previous sections, the correctness and security restrictions only make statements in the case of honestly generated signing keys. There are no guarantees for malicious keys. When an adversary maliciously creates a key pair and the public key is used to verify a signature, the adversary can manipulate the result of a verification. The result can be true or false as long as the signatures belonging to honest keys are valid. Also, due to the consistency restriction, the same verification must have the same result in a trace.

This behavior stands in contrast to the attack finding model where the result of a verification is well-defined. This additional flexibility enables the validation model to model various attacks without stating the attacks explicitly. For example, it enables an attack on the Destructive Exclusive Ownership (DEO) property, as described for signatures by Jackson et al. [17]. We translate this attack for the aggregate signature setting:

Definition 6.8 (Attack on DEO property for aggregate signatures) *The adversary is provided with an aggregate signature σ_{agg}, that validates for some messages $\boldsymbol{m}$ and public keys $\boldsymbol{pk}$. The adversary then creates a message m'_i and a public key pk'_i such that $m'_i \neq m_i$ and $pk'_i \neq pk_i$ and replaces m_i with m'_i in $\boldsymbol{m}$, resulting in $\boldsymbol{m'}$, and replacing pk_i with pk'_i in $\boldsymbol{pk}$, resulting in $\boldsymbol{pk'}$, such that* $\texttt{vfyAgg}(\sigma, \boldsymbol{m'}, \boldsymbol{pk'}) = \text{true}$.

Note that the adversary can replace multiple message and public key pairs, but they have to be of the same index. The adversary cannot, for example, replace only m_i and pk_j where $i \neq j$.

We now provide, as an example, a rule that represents the above-described attack:

```
rule DEO:
  [ !Ltk($A, ska), !Ltk($A, sk2),  Fr(~skb)
  , Fr(~ma), Fr(~mb), Fr(~m2) ]
--[ VerifyAgg(agg(<sign(~ma, ska), '1'> + <sign(~m2, sk2), '2'>),
    <~mb, pk(~skb), '1'>+<~m2, pk(sk2), '2'>, true)
  , DEO() ]->
  [ ]
```

Note that we use the `Ltk` fact for honest keys and that we use fresh values to represent the malicious keys. As this attack does not violate the computational definition of aggregate signatures, this rule can occur in a trace. It can also occur with the verification result false. In some sense, the adversary can choose the result of a verification with malicious public keys.

Wrong Formats

As stated in Section 5.1.2, we make some assumptions on the format of the validated aggregate signature and also on the indexes. To enforce those assumptions, we add additional restrictions. The restrictions are formulated on the verification facts with the verification result true. They ensure the following:

- There is a message and key pair for each aggregated signature and the indexed match.
- There is a signature for each aggregated message and key pair and the indexed match.
- The indexes inside the aggregation are distinct

- The indexes of the message and key pairs are distinct

Thus, verifications with an incorrect input format are forbidden. Examples of such wrong formats are:

- Too many signatures:

```
VerifyAgg(agg(<sign(m1, sk1), '1'> + <sign(m2, sk2), '2'>)
    , <m1, pk(sk1), '1'>, true)
```

- Non-matching indexes:

```
VerifyAgg(agg(<sign(m, sk), '1'>), <m, pk(sk), '2'>, true)
```

- Missing function agg:

```
VerifyAgg(<sign(m, sk), '1'>, <m, pk(sk), '1'>, true)
```

- Missing index:

```
VerifyAgg(agg(<sign(m, sk)>), <m, pk(sk), '1'>, true)
```

In practice, such wrong format inputs would be rejected by a verification algorithm. Thus, such verifications cannot appear in any trace. Studying this behavior further could be interesting for future work. Handling malformatted input is often not considered by symbolic models, although it plays an important role in the implementation of cryptographic primitives.

Missing Numbers

There is another special case where both the aggregation and the message and key pairs do not include an index, as shown in the following rule:

```
rule MissingNumber:
   [ Fr(~m), !Ltk($A, ~sk) ]
--[ VerifyAgg(agg(<sign(~m, ~sk)>), <~m, pk(~sk)>, true),
   MissingNumber_correctSignature() ]->
   [ ]
```

In the case of the verification result being false, the correctness restriction excludes such traces. In the case of a true verification result, however, no restriction covers it. The attempt at formulating a restriction for this special case failed. Thus, this case is treated by Tamarin as if it was a valid aggregation. We argue that this is acceptable, since the verifying agent should provide a valid input where each message and key pair includes an index. Otherwise, the verifier misuses the API and cannot expect a valid result.

6.3 Experiments

We evaluate the performance of our aggregate signature models on our example weather protocol from Section 3.1. We use the same four lemmas as for the model

	Model	
Lemma	Attack finding model	Validation model
Executable one signature (exists-trace)	1.0 s	1.4 s
Executable two signatures (exists-trace)	1.2 s	13.8 s
Message authenticity (prove)	Time out after 1h	Time out after 1h
Weak agreement (falsify)	8.5 s	23.3 s
No splitting zero attack (prove)	0.2 s	Time out after 1h
No rogue key attack (prove)	Time out after 1h	Time out after 1h

Table 6.4: Proof and trace finding times in seconds of the attack finding model and validation model, both without additional adversary capabilities. The measurements represent the CPU time of a single proof or falsification. Measured on Intel Xeon 2.20GHz 48 core with 256GiB RAM

comparison in Section 6.1.1, as well as two additional executability lemmas: one lemma for one aggregated signature and one for two aggregated signatures. The two executability lemmas prove the existence of a protocol trace without adversary interference.

We compared the proof and trace finding times with the attack finding model and the validation model. Both models do not have adversary capabilities apart from the long-term key reveal rule, stated in Section 4.3 on page 32. They correspond to the models 1 and 4 from Section 6.2.2. Table 6.4 shows the results of running each lemma for the two models. We choose a timeout of one hour, since the proofs might not terminate, due to the undecidability of the problem.

The executability lemmas, marked with the key word exists-trace, both terminate for both models. However, the trace finding takes longer for the validation model, especially for two signatures. We observe the same behavior for the falsification of lemma *weak agreement*. The models have more difficulties with the proofs. The *message authenticity* and the *no rogue key attack* proofs do not terminate for both models. However, the *no splitting zero attack* proof for the attack finding model is very fast, while it does not terminate for the validation model. Tamarin can perform this proof that fast with the attack finding model since the splitting zero attack contradicts the aggregation function. The function `vfyAgg` does not allow an aggregate signature to be aggregated once with one set of messages and then with a different set of signatures.

The non-termination probably occurs through the repeated application of rules. We constructed the protocol model such that the aggregate signature can be arbitrarily long. Thus, the aggregation and key lookup consist of the application of arbitrarily many rewriting rules, similar to the adversary aggregation rules in Section 4.2 on page 31. It is important to note that the proof time is highly dependent on the protocol. We present the results for our artificial example protocol but other protocols could behave completely differently.

6.3.1 Improve Performance

As not all proofs terminate, we evaluate some techniques to improve the proof times. We consider the following three techniques:

Limit the number of signatures: We observed that the proof finding incrementally proofs for every possible number of aggregated signatures. To prevent this infinite loop, we can add a restriction that restricts the number of aggregated signatures. With this addition, we can prove that the properties hold for this limited number of signatures. In our evaluation, we limit the number of signatures once to two and once to three.

Omit the aggregator role: As the aggregation can be done by the aggregator or by the adversary, we can omit the aggregator role. This omission simplifies the protocol and could thus help with the performance.

Model the key lookup with restrictions: Another approach is to model the key lookup with restrictions instead of using pattern matching. We replace the recursive key lookup rules with one rule, where the adversary provides the keys and we define restrictions that ensure that the provided keys match the agent names.

Table 6.5 shows the results of evaluating these three simplifications on the attack finding model and on the validation model.

Limiting the number of signatures results in terminating proofs for the attack finding and the validation models. This is not surprising as we limit the search space for Tamarin. The proof times for *weak agreement* lemma are also faster with the limited number of signatures. There is an outlier for the validation model with three signatures and the proof of the *no splitting zero* lemma, which we cannot explain. The technique of limiting the number of signatures has proven to be very effective, but it has the drawback that we can only prove the properties for this limited number of signatures. Thus we may miss some attacks. On the other hand, if an attack is possible for a small number of signatures, this simplification can be very effective for attack finding.

Omitting the aggregator role did decrease the proof times, but the same lemmas time out.

It turned out that modeling the key lookup with restrictions increased the proof times. The trace finding for the executability lemmas was increased (not shown in the table) and the falsification of the *weak agreement* lemma did not terminate in one hour.

	Message authenticity (prove)	Weak agreement (falsify)	No splitting zero attack (prove)	No rogue key attack (prove)
Attack finding model				
No simplifications	Time out after 1h	8.5 s	0.2 s	Time out after 1h
Limit number of signatures to two	2.6 s	2.8 s	0.2 s	1.0 s
Limit number of signatures to three	10.3 s	5.2 s	0.2 s	1.1 s
Omit aggregator role	Time out after 1h	3.5 s	0.2 s	Time out after 1h
Validation model				
No simplifications	Time out after 1h	23.3 s	Time out after 1h	Time out after 1h
Limit number of signatures to two	10.4 s	13.0 s	5.7 s	9.6 s
Limit number of signatures to three	70.4 s	19.6 s	293.9 s	61.2 s
Omit aggregator role	Time out after 1h	10.5 s	Time out after 1h	Time out after 1h
Model the key lookup with restrictions	Time out after 1h	Time out after 1h	Time out after 1h	Time out after 1h

Table 6.5: Proof and trace finding times in seconds for different approaches to simplify the models. The measurements represent the CPU time of a single proof or falsification. Measured on Intel Xeon 2.20GHz 48 core with 256GiB RAM

6.4 Summary

Our evaluation showed that we have the strongest assumptions on the security of the signatures in the attack finding models without additional adversary capabilities. This model is analogous to Tamarin's built-in signature model. The validation model with rogue key attack presumes the least assumptions. Thus, proofs using this model provide the strongest security guarantees. This means, the validation model can capture some attacks, that are proven to be impossible by the attack finding models. The drawback of the validation models is the increased proof time. Our evaluation showed that for both models, some proofs did not terminate. We solved this by limiting the number of aggregated signatures. This, in turn, could again result in missing attacks that are only possible for a large number of signatures. Those trade-offs between termination and capturing all possible attacks have to be considered when using our models. Note, however, that the proof and falsification times depend highly on the protocol and its model.

Chapter 7

Conclusion

In this work, we developed the first symbolic models of aggregate signatures in Tamarin and studied different approaches.

We created two new classes of aggregate signature models: attack finding models, which are close to the standard symbolic model of signatures, and validation models, which are derived from the computational definition. This second approach is inspired by Jackson et al. [17]. It extends their technique of using Tamarin's restrictions to model signatures by adding quantification over the elements in a multiset to the restrictions.

We analyzed two attacks on aggregate signatures. We showed that our models are practical for attack finding and proving security properties. With our simplification techniques, we could produce fast proofs on an artificial protocol.

Our models and the techniques we developed enable future research. Further symbolic models of multi-party signatures can be developed and our models can be improved for specific use cases. On our example protocol, some proofs and attack searches did not terminate without simplifications. The termination of proofs is highly dependent on the protocol. Thus, the performance needs to be improved for a specific protocol and should be approached when using our models in a specific protocol. This optimization is related to the problem of how to model the aggregation of arbitrarily many signatures. Our repeated application of rewrite rules results in loops, which in turn results in non-termination. Resolving this would also be beneficial for modeling further multi-party signatures.

Our models could be expanded at different points, the assumptions could be changed, more attacks could be examined or a different aggregate signature scheme could be considered. There are, however, two concrete points, that can be improved on the attack finding models: (1) As stated in Section 4.2 on page 31, an honest agent can only aggregate valid signatures and no arbitrary terms. For example, an honest agent cannot accidentally aggregate malicious signatures provided by the adversary. To capture such cases, additional rules would be needed. (2) Our colliding signature attack finding model, described in Section 4.3.2, has the limitation that only a fixed number of malicious signatures can be aggregated. Finding a method to model the aggregation of arbitrarily many malicious signatures would be interesting for future work.

In Section 5.3 on page 44, we discovered that with some restriction a proof did terminate, and with an equivalent reformulation of the restriction the proof did not terminate. It would be very interesting to evaluate this further and evaluate whether Tamarin can be optimized regarding restrictions. This could be especially beneficial for further explorations of using restrictions to model cryptographic primitives.

BLS signatures are based on bilinear pairings. Thus, it seems intuitive to use Tamarin's built-in theory for bilinear pairings. But the aggregation of BLS signatures needs the multiplication in the target group, which is not supported by Tamarin. One might be able to create an alternative bilinear pairing model that supports multiplication in the target group. The benefit of such a model would be that the attacks are closer to real-world attacks.

With this work, we began exploring how to model multi-party primitives in Tamarin. The techniques developed for our models could be used for other primitives, such as threshold signatures or group signatures. Future research in this direction could help to model various protocols.

Bibliography

[1] Specification of Ethereum 2.0 Phase 0 – The Beacon Chain. https://github.com/ethereum/eth2.0-specs.

[2] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1383–1396, New York, NY, USA, October 2018. Association for Computing Machinery.

[3] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 2.03: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf, September 2021.

[4] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In Yvo G. Desmedt, editor, *Public Key Cryptography — PKC 2003*, Lecture Notes in Computer Science, pages 31–46, Berlin, Heidelberg, 2002. Springer.

[5] Alexandra Boldyreva, Adriana Palacio, and Bogdan Warinschi. Secure Proxy Signature Schemes for Delegation of Signing Rights. *Journal of Cryptology*, 25(1):57–115, January 2012.

[6] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact Multi-signatures for Smaller Blockchains. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, volume 11273, pages 435–464. Springer International Publishing, Cham, 2018.

[7] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. Technical Report 175, 2002.

[8] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Eli Biham, editors, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656, pages 416–432. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[9] Dan Boneh, Sergey Gorbunov, Riad Wahby, Hoeteck Wee, and Zhenfei Zhang. Draft-irtf-cfrg-bls-signature-04 - BLS Signatures. https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/04/.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, Lecture Notes in Computer Science, pages 514–532, Berlin, Heidelberg, 2001. Springer.

[11] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography - Version 0.5, Jan. 2020.

[12] Chia-Network. BLS Signatures implementation. https://github.com/Chia-Network/bls-signatures, April 2021.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1773–1788, New York, NY, USA, October 2017. Association for Computing Machinery.

[14] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming*, 81(7):898–928, October 2012.

[15] Xenia Hofmeier. Tamarin theories and scrips. https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/hofmeier-aggsign.zip.

[16] Xenia Hofmeier. Tamarin theories and scrips. https://github.com/hxenia/master-thesis, October 2021.

[17] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2165–2180, London United Kingdom, November 2019. ACM.

[18] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic Protocol Analysis for the Real World. Technical Report 971, 2019.

[19] Nguyen Thoi Minh Quan. 0. Technical Report 323, 2021.

[20] Nguyen Thoi Minh Quan. Attacks and weaknesses of BLS aggregate signatures. Technical Report 377, 2021.

[21] Wahby Riad S. and Zhenfei Zhang. BLS signatures draft standard, reference implementations by Algorand. https://github.com/algorand/bls_sigs_ref, March 2021.

[22] The Tamarin Team. Tamarin Prover Manual. https://tamarin-prover.github.io/manual/index.html.