

One For All: Formally Verifying Protocols which use Aggregate Signatures (extended version)

Xenia Hofmeier¹, Andrea Raguso¹, Ralf Sasse¹, Dennis Jackson², and David Basin¹

¹*Department of Computer Science, ETH Zurich, Zurich, Switzerland*

{xenia.hofmeier, andrea.raguso, ralf.sasse, basin}@inf.ethz.ch

²*Mozilla, UK*

research@dennis-jackson.uk

Abstract—Aggregate signatures are digital signatures that compress multiple signatures from different parties into a single signature, thereby reducing storage and bandwidth requirements. BLS aggregate signatures are a popular kind of aggregate signature, deployed by Ethereum, Dfinity, and Cloudflare amongst others, currently undergoing standardization at the IETF. However, BLS aggregate signatures are difficult to use correctly, with nuanced requirements that must be carefully handled by protocol developers.

In this work, we design the first models of aggregate signatures that enable formal verification tools, such as Tamarin and ProVerif, to be applied to protocols using these signatures. We introduce general models that are based on the cryptographic security definition of generic aggregate signatures, allowing the attacker to exploit protocols where the security requirements are not satisfied. We also introduce a second family of models formalizing BLS aggregate signatures in particular. We demonstrate our approach’s practical relevance by modelling and analyzing in Tamarin a device attestation protocol called SANA. Despite SANA’s claimed correctness proof, with Tamarin we uncover undocumented assumptions that, when omitted, lead to attacks.

I. INTRODUCTION

Digital signatures are well established and widely used. Aggregate signatures are an extension of traditional digital signatures which have found favor in applications where many independent parties interact, such as the Ethereum blockchain [1], Cloudflare’s distributed random beacon [2], and Dfinity’s ‘Internet Computer’ [3]. Aggregate signatures were first introduced as a cryptographic primitive in [4] as an extension of Boneh-Lynn-Shacham (BLS) signatures [5]. BLS aggregate signatures are the most popular instantiation of aggregate signatures and they are undergoing standardization at the IETF [6].

Unfortunately, naive constructions of BLS aggregate signatures are vulnerable to signature forgeries using so-called *rogue public key attacks*, see Section II-A1, violating the basic security definition of aggregate signatures [7]. Different mitigations have been proposed to address this issue, but unfortunately none of them is suitable in all circumstances. So the current IETF draft requires implementors to choose between mitigations. This leaves protocols open to attack if implementors select the wrong mitigation, misunderstand how the mitigation must be applied, or fail to correctly implement the mitigation in their protocols. Adding further complexity,

some of the mitigations can only be applied at the protocol level, and not within the primitives themselves.

BLS signatures are also popular as single, unaggregated signatures in blockchain schemes as they are a *unique* signature scheme, meaning for each public key, there is a unique signature per message. Note that uniqueness is not guaranteed by the EUF-CMA security definition [8]. Although implementors might expect this property to extend to aggregate BLS signatures, it does not, as is demonstrated by the *splitting zero attacks* [9], see Section IV-B1. In fact, it is possible for an attacker to generate malicious public keys and an aggregate signature that will validate for any message, which trivially violates the uniqueness requirement. Perhaps surprisingly, splitting zero attacks do not violate the cryptographic security definition of aggregate signatures as no forgery has taken place against an honestly generated key, only against those keys selected by the attacker. At present, there is no mitigation for these attacks in the IETF draft.

Motivated by rogue public key attacks and splitting zero attacks, we differentiate between two classes of attacks:

Insecure behaviors: Attacks on primitives that *violate the EUF-CMA security definition*, e.g., the rogue public key attack.

Undesirable behaviors: Attacks on primitives that *satisfy the security definition* but that violate other desirable properties not captured by the security definition, such as the splitting zero attack.

The above distinction is critical in motivating our different models and understanding the kinds of attacks we find, and their scope and impact. Cryptographic primitives exhibiting insecure behaviors are insecure and should not be used without mitigations. The situation for undesirable behaviors is less straightforward. Different applications have different requirements on the primitives and thus an undesirable behavior for one application might be acceptable for another application. Thus, different primitives with different properties entail different resulting properties for the security protocols and applications using them.

Expanding on the above, we note that it is important to differentiate between attacks on primitives, including insecure and undesirable behaviors thereof, and attacks on security protocols. An adversary could exploit an attack on the primitive to attack a protocol, violating the protocol’s security properties.

However, the attack on the primitive alone might not violate the protocol’s security properties. Thus, security protocol analysis must account for the specific properties of the primitives.

With these types of attack in mind, and motivated by the complexity of the protocol-level mitigations that users of aggregate signatures must employ, we turn to building formal models of aggregate signatures suitable for carrying out automated analyses. Automated security protocol verification tools, such as ProVerif [10] and the Tamarin prover [11], are very effective in analyzing the properties of security protocols. These automated verification tools operate in the ‘symbolic model’ of cryptography, which uses abstractions of the computational definitions to enable automated attack and proof finding.

In this paper, we create the first symbolic models of aggregate signatures. Modelling multi-party primitives with an arbitrary number of signers, such as aggregate signatures, in the symbolic model is non-trivial. One challenge is finding a suitable representation of the primitive such that the primitive’s properties can be modeled without blowing up the subsequent analysis. A second challenge is formulating these properties as an equational theory, as the equations must satisfy certain well-formedness conditions to be used by Tamarin. Our models support the aggregation of arbitrarily many signatures.

We follow two approaches to modeling aggregate signatures, which results in two classes of models. Each class consists of three variants accounting for different properties of the primitive, including the behaviors discussed earlier. Table I, given in Section V-A, lists these variants.

Our first class of models is based on the computational security definition of aggregate signatures, assuming only the security of signatures that were generated from honestly generated keys. Thus, this class of models allows for undesirable behaviors such as the splitting zero attack but disallows insecure behaviors such as the rogue public key attack.

To model BLS aggregate signatures specifically, we need to consider their behaviors, such as the rogue public key attack. One of the proposed mitigation techniques specified by the IETF draft against the rogue public key attack is enforced on the protocol layer. This means that protocols using BLS aggregate signatures with this mitigation technique use an insecure primitive that, by itself, is vulnerable to the rogue public key attack, but the protocol should prevent this attack with its own additional checks. Thus, to verify such protocols, the mitigation technique’s effectiveness must be verified for each protocol where it is employed. We therefore extend our initial model with a specific adversary capability, enabling the rogue public key attack. As this model allows for all possible undesirable behaviors and even allows the insecure behavior of the rogue public key attack, one can thus prove protocol security properties under very few assumptions, thereby providing strong guarantees when successful.

However, some protocols might rely on aggregate signature primitives that guarantee additional desirable properties. Thus, we create a second family of aggregate signature models based on Tamarin’s built-in signatures. These models for-

malize security not only for honestly-generated key pairs, as the computational definition does, but also for all key pairs, honestly generated or not. These models reflect desirable behaviors such as uniqueness and thus disallow the splitting zero attack, representing primitives with more properties than required by the cryptographic security definitions.

To close the gap between our first set of models that allows for any undesirable behaviors and our second set of models that excludes all of these behaviors, we extend the second set of models to capture the splitting zero attack and the rogue public key attack. This results in a set of models that places stronger restrictions on the attacker’s capabilities than our first set of models, and these models result in different analysis times, which we evaluate in Section V-B.

We showcase our aggregate signature models on the remote device attestation protocol SANA [12], which was published at CCS’16 with a claimed security proof. SANA aims to provide scalable and secure remote attestation and uses a novel aggregate signature scheme called optimistic aggregate signatures to reduce verification overhead while minimizing the trust assumptions about the devices and the network. We modeled SANA in Tamarin and formalized the security properties stated in [12] and additional authentication properties. Our analysis revealed necessary assumptions on the device’s initialization that were not explicitly stated by SANA’s authors. Without these assumptions, Tamarin discovers attacks violating SANA’s claimed security properties. These attacks exploit flaws in the proposed authorization mechanism and result in an incorrect view of the network being presented to the party verifying the attestation. We verify, using Tamarin, the security properties under the identified assumptions.

a) Contributions: We develop the first symbolic models of aggregate signatures, implemented in Tamarin. Our models are based on the computational definition of aggregate signatures provided by Boneh et al. [4] and on BLS aggregate signatures and their IETF draft [13]. We develop two classes of models: validation models based on the computational definition of aggregate signatures and attack-finding models based on Tamarin’s built-in signatures. Additionally:

- We justify the correctness of our models by deriving them from the computational definition of aggregate signatures, which requires choosing appropriate abstractions of the computational definition. We address challenges involved with modelling the aggregation of arbitrarily many signatures by introducing different representations of aggregate signatures and techniques such as quantifying over the elements in multisets.
- Two attacks have been presented in the literature on BLS aggregate signatures: rogue public key attack and splitting zero attack. We formalize those behaviors within our attack-finding models. We show that the validation models allow for splitting zero attacks without explicitly modelling this behavior. In contrast, the rogue public key attack must be modeled explicitly as it contradicts the EUF-CMA security definition.

- We evaluate the behavior of our aggregate signature models on examples, measuring the time required to find proofs and attacks in Tamarin. As large numbers of aggregated signatures has a negative impact on Tamarin’s run time, we propose simplifications to achieve termination.
- We showcase our models in the security analysis of the remote device attestation protocol SANA [12]. We identify necessary assumptions, verify the chosen security properties under these assumptions, and present attacks that are possible without these assumptions and contradict the authors’ design requirements.

b) *Related Work*: Quan [9], [14] describes several attacks targeting the BLS signature IETF draft v4 [13]. We focus on the *splitting zero attack*, which represents an undesirable behavior of aggregate signatures. This attack motivates specifying undesirable behaviors in our aggregate signature models.

Jackson et al. [15] points out a gap between the computational definition of signatures and the standard signature model used by all symbolic analysis tools, such as Tamarin and ProVerif, going back to ProVerif’s initial publication [16]. That work only covers ‘traditional’ digital signatures and not aggregate signatures or other types of signatures. They point out behaviors of signature implementations that cannot be captured by the standard symbolic model and they define more detailed symbolic models that can capture these behaviors.

In particular, Jackson et al. define two classes of models: The first set of models extends the standard symbolic model of signatures by modelling specific behaviors with explicit additional equations and other techniques. As one cannot be sure that all possible behaviors lying in the gap between the computational and symbolic model are covered, they introduce a second approach to modeling cryptographic primitives. Instead of explicitly allowing certain behaviors, everything that does not contradict the cryptographic definition is allowed. The cryptographic definition is formulated using restrictions, which we explain in Section II-B1, restricting the allowed verification results. We follow this approach and create models of aggregate signatures using both approaches.

In the same spirit, Cremers et al. [17] create abstract models of threshold signatures using Tamarin. Their models support an arbitrary number of signers, and capture different security notions using different equational theories, restrictions, and adversary capabilities.

Le-Papin et al. [18] formalize security requirements for attestation protocols. They focus on a class of attestation protocols for a large number of devices that let a verifier identify provers with valid and invalid software. This class includes the SANA protocol, which we analyze in this work. Le-Papin et al.’s security properties include the prover’s authentication to the verifier, which we also consider in our analysis of SANA. They focus on the SIMPLE+ protocol and model it with Tamarin. For this analysis they use our aggregate signature models from the unpublished master thesis [19]. They chose the attack-finding models, as they are faster.

c) *Outline*: In Section II, we provide background on aggregate signatures and the Tamarin prover. In Section III, we

present our validation models, which follow the computational definition of aggregate signatures and we extend them for the rogue public key attack. In Section IV, we present our attack-finding models, based on the standard signature model and extensions for colliding signatures and rogue public key attacks. We evaluate these models in Section V by comparing them on a simple protocol and comparing their proof times. In Section VI, we showcase our models on a security analysis of SANA and, in Section VII, we draw conclusions.

II. BACKGROUND

In this section, we provide background on aggregate signatures and the Tamarin prover.

A. Aggregate Signatures

Aggregate signature schemes [4] compress multiple signatures $\sigma = (\sigma_1, \dots, \sigma_n)$ of different messages into one short aggregate signature σ_{agg} . This compression reduces storage and bandwidth requirements, and is useful in situations where many independent signers interact.

An *aggregate signature scheme* is a signature scheme with the two additional algorithms, agg and vfyAgg . The *aggregation algorithm*’s agg input is a vector of public keys $\mathbf{pk} = (pk_1, \dots, pk_n)$ and a vector of signatures $\sigma = (\sigma_1, \dots, \sigma_n)$ and it outputs an aggregate signature σ_{agg} . The *deterministic aggregate verification algorithm*’s vfyAgg input is a vector of public keys $\mathbf{pk} = (pk_1, \dots, pk_n)$, a vector of messages $\mathbf{m} = (m_1, \dots, m_n)$, and an aggregate signature σ_{agg} . It outputs either true or false. The scheme is *correct* if for all \mathbf{pk} , \mathbf{m} , and σ , if $\text{vfy}(pk_i, m_i, \sigma_i) = \text{true}$ for $1 \leq i \leq n$, then $\text{Pr}[\text{vfyAgg}(\text{agg}(\mathbf{pk}, \sigma), \mathbf{m}, \mathbf{pk}) = \text{true}] = 1$. We provide a more formal definition in Appendix A, Definition 1. Signatures can be aggregated without knowledge of the signing keys. Thus, any agent can aggregate the signatures that it possesses.

Aggregate signatures have a well-established security definition [7], which is the natural extension of *existential unforgeability under a chosen message attack (EUF-CMA)* to the many-signer case. Here, we describe the attack game and we provide the detailed definition in Appendix A, Definition 2.

The challenger generates a key pair $(pk, sk) \leftarrow \text{gen}()$ and sends pk to the adversary. The adversary sends the challenger signing queries, each consisting of a message $m^{(i)}$. The challenger responds to each signing query with $\sigma^{(i)} \leftarrow \text{sign}(m^{(i)}, sk)$. Eventually, the adversary outputs a candidate aggregate forgery $(\mathbf{pk}, \mathbf{m}, \sigma_{\text{agg}})$, where $\mathbf{pk} = (pk_1, \dots, pk_n)$ and $\mathbf{m} = (m_1, \dots, m_n)$. The adversary wins the game if $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}) = \text{true}$ and one of the public keys $pk_j \in \mathbf{pk}$ is the public key pk generated by the challenger where the adversary did not issue a signing query for the corresponding message m_j .

1) *BLS Aggregate Signatures*: Boneh–Lynn–Shacham (BLS) aggregate signatures [4] were the first instantiation of aggregate signatures. These signatures are used in various applications, resulting in their standardization in an IETF draft [6]. We provide a full description of BLS signatures

and their properties in Appendix B, but introduce their key properties below.

Naive constructions of BLS aggregate signatures are vulnerable to the *rogue public key attack*. This enables an attacker, through a malicious choice of their public key, to forge an aggregate signature that verifies under the victim’s and the attacker’s public keys. See Appendix B, Section B-C, for details. The IETF draft [6] defines three mitigations to prevent this:

- 1) Reject duplicate messages during the verification.
- 2) Bind signed messages to their public keys.
- 3) Require signers to prove possession of their secret key.

These techniques have different trade-offs and are thus suited for different applications. The first two techniques are part of the verification and signing algorithm and thus they are part of the primitive. However, the third technique requires additional checks to be carried out by the protocol.

In contrast to the aggregate signature definition, Appendix A, Definition 1, the aggregation of BLS signatures does not require the public keys. As the aggregation does not require one to validate the aggregated signatures, we also omit the public keys in our models.

B. The Tamarin Prover

The Tamarin Prover [11], [20] is an automated verification tool used to model and analyze security protocols. Given a protocol model and a security property, Tamarin either constructs a proof or finds an attack on the property. As the underlying problem is undecidable, the tool may not always terminate.

Tamarin operates in the symbolic model of cryptography where cryptographic messages are represented as terms. Each cryptographic message is a constant, a variable, or a function symbol applied to some messages. The function symbols represent cryptographic primitives or grouping operations. The properties of these functions are expressed with equational theories. The equations model, for example, that decrypting an encrypted ciphertext, using the appropriate decryption key, yields the original plaintext, as expected. Tamarin supports various built-in equational theories for cryptographic primitives, such as for signing, which we introduce shortly. Tamarin also supports user-defined function symbols and equations, which must satisfy certain well-formedness conditions [21].

Tamarin works with a model of a standard Dolev-Yao adversary that controls the network. Thus, the adversary learns any messages sent over the network and can derive messages from its knowledge. However, cryptography is assumed to work perfectly. This means, for example, that for decryption the adversary needs the appropriate decryption key, and importantly it has no cryptanalytic capabilities. Hence, given a cipher text, the adversary learns the plaintext if and only if it possesses the associated decryption key. The adversary can also apply functions to messages in its knowledge, for example encrypting messages with keys it knows.

Tamarin models are specified using terms, facts, and rules. We already presented the terms that represent messages. Facts are of the form $F(t_1, \dots, t_k)$, where F is a fact symbol

with some fixed arity and t_i are terms. Fact symbols are used to represent protocol participant states, messages on the network, the adversary’s knowledge, and the generation of fresh nonces. Fact names are user-chosen and have no intrinsic meaning, with the exception of some special built-in names we present next. Sending a message m to the network is represented by the fact $\text{Out}(m)$ and receiving a message m is represented by the fact $\text{In}(m)$. The generation of a fresh nonce $\sim n$ is represented by $\text{Fr}(\sim n)$.

A labeled multiset rewrite rule consists of the keyword `rule` followed by the rule name and three multisets of facts: the premises, the actions (also called labels), and the conclusions. The following rule, called `Example`, has its premise on line 2, action on line 3, and conclusions on line 4.

```
1 rule Example:
2   [ Fr(~m) ]
3 --[ Label(~m) ]->
4   [ State(~m), Out(~m) ]
```

In this rule, an agent creates a fresh nonce $\sim m$, saves it in the agent’s state and sends this nonce to the network. The rule is labeled with the action fact $\text{Label}(\sim m)$.

Tamarin considers arbitrarily many protocol executions interleaved in parallel. These executions, also capturing the adversary’s behavior, are represented as a labeled transition system. The protocol’s state is represented by a multiset of facts representing the local states of the protocol participants, the messages on the network, and the adversary’s knowledge. Labeled multiset rewrite rules model both the protocol steps and the adversary capabilities. The protocol can transition into a new state by applying such a rule to the protocol state. A rule can be applied if there is a subset of facts in the current state that matches the rule’s premises. Applying the rule removes this matched subset of facts from the state and adds instantiations of the rule’s conclusions under the matching substitution.

A protocol’s execution is represented by the repeated application of multiset rewrite rules to the protocol’s state, where the initial state is the empty multiset. The trace of such an execution is defined by the sequence of the instantiated action facts of the rewrite rules.

The semantics of a security protocol is defined as the set of all traces of the protocol’s labeled transition system. We express the protocol’s security properties as trace properties, which are also sets of traces. Hence, a protocol satisfies a security property if the protocol’s set of traces is a subset of the property’s traces. If this is not the case, counter examples (i.e., attacks) exist that violate the property.

In Tamarin, we formulate security properties as first-order formulas, called *lemmas*. There are two kinds of lemmas: the most common one expresses that the property must hold for all protocol traces. Alternatively, *executability lemmas*, marked with the keyword `exists-trace`, state that there exists a trace for which the property holds. The first kind of lemma states that a property holds for all protocol behaviors. The second kind is mostly used to check that the protocol model is executable, increasing one’s confidence in the given model.

The set of traces considered by Tamarin can be restricted using *restrictions*, which are first-order formulas, with a syntax similar to lemmas. Restrictions express properties that the traces considered by Tamarin must have. They can be used, for example, to enforce that certain rules are only applied once, or to formalize signature verification.

1) *Signatures in Tamarin*: Tamarin provides the following built-in equational theory for signatures.

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

This theory provides four function symbols for signing, verification, deriving a public key from a secret key, and `true`, which represents the verification result. The equation expresses that signatures verify (they equal `true`) exactly if the message and public key match the signed message and signing key.

This is a coarse abstraction of the computational definition as noted in [15]. There, the correctness property gives guarantees only for honestly-generated key pairs and the EUF-CMA security definition guarantees that all efficient adversaries have a negligible advantage of winning the attack game for honestly-generated keys. The computational definition provides no guarantees for signatures with not honestly-generated keys. Prior symbolic models guarantee correctness and security even for not honestly-generated keys. We now refer to honestly-generated keys and signatures using these keys as *honest keys* and *honest signatures*. Keys and signatures that are not honestly generated are called *non-honest keys* and *signatures*.

III. VALIDATION MODELS

In this section, we describe our validation models and in the next section we describe our attack-finding models. All models are available at [22]. Moreover, for additional details, see [19].

Our validation models are based on the computational definition of aggregate signatures. Here, we represent the verification of an aggregate signature σ_{agg} using predicate notation: $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, b)$, where b is the verification result. In these models, every behavior that is not explicitly forbidden by the definition is possible. Hence each verification $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, b)$ functions as an adversary controlled oracle whose verification result b could be either true or false¹ unless governed by a restriction.

Our validation models rely on the computational definition of aggregate signatures, namely on their correctness, consistency, and on EUF-CMA security or unforgeability. We say that a signature or aggregate signature *verifies* if the verification algorithm outputs true. The *correctness* of aggregate signatures states that if all signatures σ_i in an aggregation $\text{agg}(\sigma_1, \dots, \sigma_n)$ verify, the aggregation must verify as well. *Unforgeability* states that if an aggregate signature verifies, all aggregated signatures σ_i either verify or the public key pk_i is not honest. *Consistency* states that the verification algorithm vfyAgg is deterministic, meaning a given verification must always return the same result. We model the verification of

aggregate signatures with these three properties: correctness, unforgeability, and consistency. Next, we formalize these properties as restrictions.

A. Formalizing the Properties

The correctness definition of aggregate signatures, Appendix A, Definition 1, states that if all signatures in an aggregation verify, the aggregation must verify as well. We combine this definition with the correctness definition of signatures stating that for an honestly-generated key pair (sk, pk) , the signature $\sigma = \text{sign}(m, sk)$ must verify for message m and public key pk . This results in the following restriction, where $\text{Honest}(pk)$ states that the public key pk was honestly generated and sk_j is the secret key corresponding to the public key pk_j .

Restriction 1 (Correctness of aggregate signatures).

$$\forall \mathbf{pk}, \mathbf{m}, \sigma. (\forall i \in \{1, \dots, n\}. (\text{Honest}(pk_i) \wedge \sigma_i = \text{sign}(m_i, sk_i)) \Rightarrow \text{vfyAgg}(\text{agg}(\sigma), \mathbf{m}, \mathbf{pk}, \text{true}))$$

The EUF-CMA security definition of aggregate signatures, Appendix A, Definition 2, states that no adversary can win the attack game with a non-negligible probability. We abstract this definition for the symbolic model 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 $(\mathbf{pk}, \mathbf{m}, \sigma_{\text{agg}})$ where $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, \text{true})$, where one public key $pk_j \in \mathbf{pk}$ is honestly generated, the corresponding secret key sk_j is not known to the adversary, and the corresponding message $m_j \in \mathbf{m}$ was not queried. As we assume that such a forgery is not possible, for each triple $(\mathbf{pk}, \mathbf{m}, \sigma_{\text{agg}})$, each public key $pk_i \in \mathbf{pk}$ is either not honest or there is a corresponding signature σ_i that is honestly created, which means that $\sigma_i = \text{sign}(m_i, sk_i)$. Note that σ_{agg} can be an arbitrary term if all public keys are not honest. Only if some public keys are honest, must it be an aggregation of signatures $\sigma_{\text{agg}} = \text{agg}(\sigma)$. We express this as the following property:

$$\begin{aligned} &\forall \mathbf{pk}, \mathbf{m}, \sigma_{\text{agg}}. \text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, \text{true}) \\ &\quad \Rightarrow (\forall i \in \{1, \dots, n\}. \neg \text{Honest}(pk_i) \\ &\quad \vee (\exists \sigma. \sigma_{\text{agg}} = \text{agg}(\sigma) \wedge (\forall i \in \{1, \dots, n\}. \neg \text{Honest}(pk_i) \\ &\quad \vee \exists \sigma_j \in \sigma. \sigma_j = \text{sign}(m_i, sk_i))))). \quad (1) \end{aligned}$$

To improve Tamarin's termination, we simplify this restriction by assuming the following:

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

We justify these assumptions with input validations that are performed in practice. For example, the IETF draft for BLS signatures [6] requires the public key and message vectors to be of the same length. We enforce the above assumptions

¹For our validation models, we introduce the nullary function `false`.

with some additional restrictions. With these assumptions, we can simplify Property 1 and get the following unforgeability restriction.

Restriction 2 (Unforgeability of aggregate signatures).

$$\forall \mathbf{pk}, \mathbf{m}, \sigma. \text{vfyAgg}(\text{agg}(\sigma), \mathbf{m}, \mathbf{pk}, \text{true}) \\ \Rightarrow (\forall i \in \{1, \dots, n\}. (\neg \text{Honest}(pk_i) \vee \sigma_i = \text{sign}(m_i, sk_i)))$$

The verification algorithm's deterministic behavior is expressed as the following consistency restriction.

Restriction 3 (Consistency of aggregate signatures).

$$\forall \mathbf{pk}, \mathbf{m}, \sigma_{\text{agg}}, b_1, b_2. \text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, b_1) \\ \wedge \text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, b_2) \Rightarrow b_1 = b_2$$

Usually, we only model honest keys in Tamarin as standard primitives and do not differentiate between honest and non-honest keys. However in our restriction-based models, correctness and unforgeability restrictions only specify the verification results for honest keys. Signature aggregations of non-honest keys can verify arbitrarily, while fulfilling the consistency restriction. We thus add registration rules of non-honest keys for the adversary. This enables various undesirable behaviors, such as the splitting zero attack, which we will discuss in Section IV-B1. In Section V-A, we will compare our validation model without non-honest keys, which does not allow undesirable behaviors with non-honest keys, to the validation model with non-honest keys, which allows such behaviors.

B. Signature Aggregation in Tamarin

We represent signature aggregations by the function $\text{agg}/1$ applied to a multiset of signatures. As the correctness and unforgeability restriction require accessing signatures, messages, and public keys of the same index, we introduce explicit indices. For example, we model $\text{agg}(\text{sign}(m_1, pk_1), \text{sign}(m_2, pk_2))$ as

$$\text{agg}(\langle \text{sign}(m_1, pk_1), \text{ind}_1 \rangle + \langle \text{sign}(m_2, pk_2), \text{ind}_2 \rangle),$$

where multisets are represented by the associative-commutative operator $+$ and tuples containing the terms a and b are represented by $\langle a, b \rangle$. The indices $\text{ind}_1, \dots, \text{ind}_n$ can be modeled in Tamarin in various ways, such as fresh values, constants, counters, or values provided by the adversary. Each option has different trade-offs. We opted for fresh values as they are always distinct. However, they must be provided to the adversary after aggregation, as indices should be public.

To model the validation of a signature aggregation, the action fact VfyAgg is added to the appropriate agent rule. The arguments are the signature aggregation, a multiset of triples of message, public key, and index, and the expected verification result. For example, the validation of the signature aggregation agg on messages m_1, m_2 and public keys pk_1, pk_2 , with the result true , is represented by the following action fact:

$$\text{VfyAgg}(\text{agg}, \langle m_1, pk_1, \text{ind}_1 \rangle + \langle m_2, pk_2, \text{ind}_2 \rangle, \text{true}).$$

The abstract restrictions described in Section III-A must be adapted for Tamarin. This includes case distinctions for one or multiple aggregated signatures and translating the quantification over the indices into Tamarin's syntax. The latter utilizes the associative-commutative property of Tamarin's multisets. Namely, the quantification over all pairs in a multiset can be expressed as $\text{All } a \ b \ r. \langle a, b \rangle + r$. Note that this quantification requires the multiset's elements to be identifiable by an applied function such as a pair. As otherwise pattern matching for an arbitrary term x in $x + y$, both x and y could be single elements or multisets themselves. See Appendix C for more details on formalizing the restrictions in Tamarin.

C. Rogue Public Key Attack

The validation models discussed so far are based on the EUF-CMA security definition of aggregate signatures and include no further desirable properties. Thus, as mentioned, the validation model with non-honest key registration permits undesirable behaviors that conform with our restrictions, such as the splitting zero attack. However, the rogue public key attack, introduced in Section II-A1, violates the security definition and is thus not permitted by this model. As mentioned, one of the mitigation techniques against the rogue public key attack, Proof of Possession, relies on the protocol layer. To verify the effectiveness of such protocol-based mechanisms, we require models of aggregate signatures that are vulnerable to the rogue public key attack. Thus, we extend our validation models to model BLS aggregate signatures with a protocol-based mitigation by adding an adversary capability for a rogue public key attack.

First, we provide a high-level description of the rogue public key attack and then present our Tamarin model extension. See Appendix B, Section B-C for a detailed description of the rogue public key attack.

The rogue public key attack allows an adversary to calculate a rogue public key pk_{rogue} from a target public key pk_{target} without being able to calculate the corresponding rogue secret key. However, the adversary can create a rogue signature aggregation $\sigma_{\text{agg}_{\text{rogue}}}$ that is valid for a chosen message m repeated twice, the target public key pk_{target} and the rogue public key pk_{rogue} :

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

Thus, the adversary creates a forgery for the target public key.

We enable the adversary to create rogue signature aggregations with additional adversary rules, see [19] for details. The first rule creates a rogue public key from a target public key and the other rules create a rogue signature aggregation from a rogue public key. The rogue signature aggregation is represented as a regular signature aggregation that can be verified as any other signature aggregation. Thus, the aggregation contains a forged signature of the target key and a signature of the rogue key. This second signature therefore contains the rogue secret key. Note, in practice, that the adversary creates the rogue public key without the secret key. Thus in our model, the adversary must not learn the rogue secret key. We therefore

use special *private* functions that the adversary cannot apply, which can only be used in rules. We introduce the private function `rogueSk/1` to represent the rogue secret key of some target public key.

The first added adversary rule provides the rogue public key `pk(rogueSk(pk_target))` for a given target key `pk_target` and registers the rogue public key in the public key infrastructure (PKI). The second added rule provides the rogue signature aggregation for a rogue public key `pk(rogueSk(pk(skTarget)))` and a message `m`. Note that it does not require the knowledge of the honest signature `sign(m, skTarget)`. The returned rogue signature aggregation has the form

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

Note that this rule extracts the target secret key `skTarget` and the rogue secret key `rogueSk(pk(skTarget))` to create the rogue signature aggregation, while the adversary cannot access either of the two secret keys. A third adversary rule aggregates additional signatures to the rogue aggregation.

IV. ATTACK-FINDING MODELS

The validation models presented so far only guarantee unforgeability, correctness, and consistency, but no further desirable properties (see Section I). As the unforgeability and correctness restrictions only define the behavior of aggregate signatures with honestly-generated keys, the behavior of non-honest keys can be arbitrary and indeed these models allow for many undesirable behaviors with non-honest keys. This supports proving security properties of protocols under very few assumptions, which provides strong guarantees. However, some security protocols might rely on primitives with stronger guarantees. Our second family of models, the attack-finding models described in this section, forbids any undesirable behaviors with non-honest keys by guaranteeing unforgeability and correctness not only for honestly-generated key pairs, but for all key pairs.

Our attack-finding models of aggregate signatures are based on Tamarin’s built-in signature model, described in Section II-B1. We define

$$\text{vfyAgg}(\sigma_{\text{agg}}, (m_1, \dots, m_n), (\text{pk}(sk_1), \dots, \text{pk}(sk_n))) = \text{true}$$

if and only if $\sigma_{\text{agg}} = \text{agg}(\sigma_1, \dots, \sigma_n)$, where $\sigma_i = \text{sign}(m_i, sk_i)$ for all $1 \leq i \leq n$. Note that this definition considers all key pairs, honest or not. This is an abstraction of the EUF-CMA security definition of aggregate signatures, Appendix A, Definition 3, which only considers honestly-generated keys.

A naive approach to defining `vfyAgg` with recursive equations fails due to Tamarin’s requirements on user-defined equations. Thus, we model the verification with a single, non-recursive equation that checks that the messages and keys of the aggregated signatures match the messages and keys provided for the verification.

```
1 equations:
2 vfyAgg(agg(m_list, pk_list), m_list, pk_list) = true
```

We represent an aggregate signature as the lists of messages `m_list` and public keys `pk_list` of the aggregated signatures. These lists can be modeled using Tamarin’s built-in tuples or multisets. We choose tuples to more closely resemble the vectors from the computational definition. We represent a signature aggregation `agg(sign(m_1, sk_1), ..., sign(m_n, sk_n))` by

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

Note that in Tamarin’s symbolic model of signatures, public keys can be directly derived from secret keys. Thus representing an aggregate signature using public keys only requires the application of the `pk` function. Using public keys instead of secret keys allows for direct comparison in the above equation.

Agents aggregating signatures must extract the signed messages and signing keys from the signatures. The following example shows a rule where an agent aggregates the two signatures `sign(m1, sk1)` and `sign(m2, sk2)`.

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

Note that this rule implicitly verifies the signatures using pattern matching.

To prevent the adversary from directly creating an aggregate signature from a list of public keys and messages without knowing the signatures, the function `agg` is private. Thus, the adversary cannot apply `agg`, but agents aggregating signatures can be modeled as in the above rule.

In practice, anyone can aggregate signatures, also the adversary. Thus, we provide adversary rules for incremental signature aggregation. Note that not all aggregate signature schemes offer incremental aggregation and our models could be adapted for non-incremental aggregation. Also note that because Tamarin’s tuples are not associative, the adversary aggregation rules and the agent aggregation rules must produce and accept aggregations of the same shape, namely listing the messages and public keys as `<a1, <a2, ..., <an-1, an>...>` and not as `<...<a1, a2>, ..., an-1>, an>`.

The adversary aggregation rules only allow the aggregation of valid signatures. Thus, the resulting aggregate signature is valid by construction. An invalid aggregation can be modeled by an arbitrary `term`. The verification of this arbitrary term will not validate, as the verification equation cannot be applied to `vfyAgg(term, m, k)`.

So far, we created an abstract model of aggregate signatures which behaves ‘perfectly’. In the next two subsections, we extend this model to model BLS signatures by modeling their real-world behavior and known attacks.

A. Rogue Public Key Attack

As discussed in Section III-C, some versions of BLS aggregate signatures mitigate the rogue public key attack on the protocol layer. We thus add an adversary capability for the rogue public key attack.

We extend our model by adding an adversary rule, similar to the validation model with the rogue public key attack

described in Section III-C. In contrast to the validation model extension, we do not model the rogue secret key but only the rogue public key. We model rogue public keys using the function `roguePk/1` applied to the target public key. The adversary can register these rogue public keys in the PKI with an additional rule.

We differentiate between honest signature aggregations and rogue signature aggregations by renaming the function `agg/2` to `validAgg/2` and introducing the new (also private) function `rogueAgg/2`. Differentiating between valid and rogue signature aggregations makes reasoning about attacks easier, but it is not necessary. Both aggregations could also be modeled using one function symbol. The verification of valid signature aggregations and rogue signature aggregations behave the same. This is modeled by an additional equation for the verification of rogue signature aggregations. The adversary creates rogue signature aggregations using an additional aggregation rule.

```

1 rule Adv_RogueKey_Aggregation_new:
2 [In(<m, roguePk(pkTarget)>) -->
3 [Out(rogueAgg(<m,m>, <pkTarget,roguePk(pkTarget)>))]

```

This rule creates a rogue signature aggregation for a target public key and a corresponding rogue public key. Additional rules allow the adversary to further aggregate combinations of rogue signature aggregations, valid signature aggregations, and valid signatures.

B. Colliding Signatures

Our standard attack-finding model guarantees unforgeability and correctness for all keys, honestly generated or not. Thus, this model guarantees desirable properties such as uniqueness, which states that for each vector of public keys, there is a unique aggregate signature for every message vector. Uniqueness holds for single BLS signatures. However, uniqueness fails for BLS *aggregate* signatures, as demonstrated by the splitting zero attack. To more closely model BLS aggregate signatures, we extend this model to capture this attack.

We first describe this attack at a high level. We then provide a more general definition of colliding signatures that includes the splitting zero attack and we extend our model for colliding signatures. See Appendix B for more details on BLS aggregate signatures and the splitting zero attack.

1) *Splitting Zero Attack*: Quan [9], [14] describes several attacks on BLS aggregate signatures, with a focus on the IETF draft v4 [13]. We focus here on the splitting zero attack.

Quan observed that for single BLS signatures, any signature signed with a non-honest key $sk_{adv} = 0$ verifies for any message. As the corresponding public key is the identity element, this attack can be easily detected, as done by the BLS IETF draft [13], with a check in the verification algorithm.

Quan [9] extends this attack for BLS aggregate signatures and calls it the *splitting zero attack*. The attack bypasses the identity element check by using multiple non-honest keys. The adversary chooses two non-honest keys, such that $sk_{mal_1} + sk_{mal_2} = 0$ and sk_{mal_1} and sk_{mal_2} are each non-zero. Thus, the corresponding public keys will

validate. However, aggregating non-honest signatures using these non-honest keys $sign(m, sk_{mal_i})$ and honest signatures σ results in an aggregate signature that verifies for different message vectors. Namely, the aggregate signature $agg(sign(m, sk_{mal_1}), sign(m, sk_{mal_2}), \sigma)$ validates against the message vector (m', m', \mathbf{m}) , where \mathbf{m} is the message vector signed by σ and m' can be any message and not the signed message m as expected.

This attack does not violate the EUF-CMA security definition of aggregate signatures provided in Appendix A, Definition 3, as it creates a non-honest signature for a non-honest key and not a signature forgery for an honest key. It can still be an unexpected and undesirable behavior for a protocol designer.

2) *Extending the model*: We generalize the splitting zero attack as colliding signatures. We define colliding signatures of aggregate signatures, analogous to colliding signatures of single signatures according to Jackson et al. [15], as follows: For non-honest keys $sk_{mal_1}, \dots, sk_{mal_k}$, messages $m_1, \dots, m_k, m_l, \dots, m_n$, honest keys sk_l, \dots, sk_n , and honest signatures $\sigma_l = sign(m_l, sk_l), \dots, \sigma_n = sign(m_n, sk_n)$, the aggregation

$$\sigma_{agg_{mal}} = agg(\sigma_{mal_1}, \dots, \sigma_{mal_k}, \sigma_l, \dots, \sigma_n), \quad (2)$$

where $\sigma_{mal_i} = sign(m_i, sk_{mal_i})$, will validate against any messages m'_1, \dots, m'_k and the messages m_l, \dots, m_n :

$$vfyAgg(\sigma_{agg_{mal}}, (m'_1, \dots, m'_k, m_l, \dots, m_n), (pk_{mal_1}, \dots, pk_{mal_k}, pk(sk_l), \dots, pk(sk_n))) = true. \quad (3)$$

With these colliding signatures, we can capture the splitting zero attack, which is a special case of a colliding signature where $m_1 = m_2 = \dots = m_k$ and $m'_1 = m'_2 = \dots = m'_k$.

We model colliding signature aggregations with the private function `zeroAgg/2`. The first argument is a valid signature aggregation and the second is a list of non-honest public keys. The colliding signature aggregation from Equation 2 is modeled as:

```

zeroAgg(validAgg(<m1, ..., mn>, <pk(sk1), ..., pk(skN)>)
, <pk(skMal1), ..., pk(skMalK)>).

```

The verification equations only check the messages and public keys of the honest signatures, and the non-honest public keys are ignored. Each verification equation covers a fixed number of non-honest signatures. Thus, our model cannot cover arbitrarily many non-honest signatures. The following equation, for example, models the case of one non-honest key.

```

1 equations: vfyAgg(zeroAgg(validAgg(m, k), kZero)
2 , <m_non_honest, m>, <kZero, k>) = true

```

The registration of non-honest keys and the aggregation of colliding signatures by the adversary is modeled similarly to the rogue public key attack described in Section IV-A.

V. EVALUATION

In this section, we provide guidance on model selection and identify which models should be used for which modelling scenarios. We have also carefully checked that the verification of different aggregate signatures behaves as expected. We do

not report on that in detail, but rather refer the reader to [19] and the experiments at [22].

We first examine the differences between the attack-finding and validation models, as well as between the different versions of these models, by comparing them on a simple protocol. We also evaluate the proof times and attack-finding times on that protocol, illustrating the termination issues we encountered. Together, this provides the modeler with some intuition on which models should be used for their verification efforts.

A. Comparing the Models

The validation models allow for arbitrary undesirable behaviors by providing the adversary with non-honest keys. In contrast, the standard attack-finding model does not allow for any undesirable behaviors and thus models a primitive with much stronger properties. Thus, proofs with the validation models provide stronger guarantees than proofs with the attack-finding models. However, there are also reasons to use the attack-finding models. First, protocols might rely on primitives with stronger guarantees than the primitives modeled by the validation models. Second, as we will see in Section V-B, achieving termination with the verification models is more difficult than with the attack-finding models. Thus, weaker proof guarantees can be chosen over non-termination. And finally, as the name *attack-finding model* suggests, these models can be used to identify the behavior of the signature that allows for the attack.

Our extensions for the rogue public key attack are intended to model BLS aggregate signatures that mitigate the rogue public key attack on the protocol layer, such as Proof of Possession. The other mitigation techniques mitigate the attack as part of the primitive's design and thus the mitigation can be assumed to be effective.

As mentioned, we highlight the differences of our attack-finding models and validation models with a simple protocol. Signers sign fresh messages with their secret keys and send these messages and signatures to an aggregator who aggregates an arbitrary number of these signatures and sends the signature aggregation to the verifier who looks up the signers' keys in the PKI and verifies the aggregate signature. We compare the model's properties using four lemmas:

Message authenticity For each message m , signed with a key of an honest signer S and received by the honest verifier V , S must have signed m .

Weak agreement As defined by Lowe [23], this property holds when the verifier V verifies a signature for a signer S , then S intended the message for V . This property does not hold trivially, as the signer does not include any information on the verifier in the signature.

No splitting zero attack An aggregate signature σ_{agg} cannot be validated twice using different messages \mathbf{m}_a and \mathbf{m}_b .

No rogue key attack An honest verifier cannot verify an aggregate signature for a message m and the public key of an honest signer S , when S did not sign m .

We summarize the results in Table I and refer to the Models 1-6 based on the numbers assigned in this table.

TABLE I: Proven and falsified lemmas for all models

Model		Lemma			
		Message authenticity	Weak Agreement	No splitting zero attack	No rogue key attack
Validation models					
1	No non-honest keys	Proven	Falsified	Proven	Proven
2	With non-honest keys	Proven	Falsified	Falsified	Proven
3	Rogue public key	Falsified	Falsified	Falsified	Falsified
Attack-finding models					
4	No adversary capabilities	Proven	Falsified	Proven	Proven
5	Colliding signatures	Proven	Falsified	Falsified	Proven
6	Rogue public key	Falsified	Falsified	Proven	Falsified

We compare the attack-finding Model 4 without additional adversary capabilities, as well as the two attack-finding Models 5 and 6 that support colliding signatures and rogue public key attacks with three versions of the validation model: the validation Models 1 and 2 with and without non-honest keys and the validation Model 3 supporting the rogue public key attack. Note that these results come from simplified models due to termination issues, see Section V-B for details.

The validation Model 3 with rogue public key attacks provides the weakest properties with respect to our lemmas. And the attack-finding Model 4 and validation Model 1 without non-honest keys provides the strongest properties with respect to our lemmas. As mentioned above, the undesirable behaviors of the validation models are allowed by providing the adversary with non-honest keys. Thus, the validation Model 1 without non-honest keys does not allow for colliding signatures and thus does not violate the *no splitting zero attack* lemma, whereas the validation Model 2 with non-honest keys allows for this undesirable behavior and violates the lemma.

The attack-finding Model 5 with colliding signatures and the validation Model 2 with non-honest keys have the same verification results with respect to our lemmas. What the table does not show is that the validation Model 2 covers various attacks without stating them explicitly. For example, there is an attack on the Destructive Exclusive Ownership (DEO) property, known from [24], which is possible with the validation Model 2 but not with the attack-finding Model 5. In other words, we can formulate additional properties that are falsified for Model 2 and are proven for Model 5.

B. Experiments

In this section, we evaluate how well our models can be used for verification and falsification in practice, with reasonable analysis times. We compare the basic validation Model 1 and the basic attack-finding Model 4, both without additional adversary capabilities as they have the same verification results with respect to our lemmas; this allows us to compare their verification and falsification times with respect to each lemma. We evaluate the models on the above simple protocol and lemmas. We present the results in Table II.

TABLE II: Proof and trace finding times in seconds for the simplified and unsimplified models. The measurements represent the CPU time of a single proof or falsification. Measurements made with Tamarin’s development version compiled at 05.03.2021 on an Intel Xeon 2.20GHz 48 core computer with 256GiB RAM. The time out is set to one hour.

		Executable two signatures (exists-trace)	Message authenticity (prove)	Weak agreement (falsify)	No splitting zero attack (prove)	No rogue key attack (prove)
1	Validation model					
1a	No simplifications	13.8 s	Time out	23.3 s	Time out	Time out
1b	Limit number of signatures to three	14.3 s	1.2 min	19.6 s	4.9 min	1.0 min
4	Attack-finding model					
4a	No simplifications	1.2 s	Time out	8.5 s	0.2 s	Time out
4b	Limit number of signatures to three	1.2 s	10.3 s	5.2 s	0.2 s	1.1 s

Our original models, presented in the Rows 1a and 4a, did not terminate for the proof search for the *message authenticity* and *no rogue key attack* lemmas. The non-termination probably stems from the repeated application of rules. These rules enable arbitrarily many aggregated signatures by modeling the aggregation and key lookup using loops, similar to the aggregation rules in Section IV.

We apply and evaluate different methods to improve proof times. Here we discuss the most effective method, which is limiting the number of aggregated signatures with restrictions. For more details see [19]. Table II shows the results of evaluating these simplifications on the validation Model 1b, and the attack-finding Model 4b. Limiting the number of aggregated signatures directly limits Tamarin’s search space as aggregating signatures beyond the stated limit is not considered. Thus, it is not surprising that with this method the proofs of all the previously non-terminating lemmas terminate. We also presented these results in Section V-A. The drawback of this technique is that we can only prove properties for this limited number of signatures and we may miss attacks when more signatures are aggregated than was analyzed. However, if an attack is possible for a small number of signatures, this simplification can be very effective for finding attacks.

The fundamental difference between the attack-finding models and validation models is that the attack-finding models explicitly model the allowed verifications while the validation models disallow unwanted behaviors. This should make the search space for the validation models larger than the search space for the attack-finding models. This is confirmed by our experiments. Note that Tamarin can prove the *no splitting zero attack* lemma in under 0.2 seconds for the attack-finding Model 4a since this attack directly contradicts the aggregation equation.

VI. CASE STUDY: SANA

In the previous section, we evaluated our aggregate signature models on a simple, artificial protocol. In this section, we demonstrate that our models can be used to analyze substantially larger and more complex protocols.

We analyze the Scalable Aggregate Network Attestation (SANA) protocol [12]. SANA attests whether devices’ software state is good, i.e. it matches the latest non-compromised software version, particularly when many devices are on the same network and are expected to be in identical states.

This is helpful in contexts like building automation and IoT. There, devices could be responsible for access control, e.g., limiting the access of unauthorized entities to the building. If the attestation protocol is insecure, compromised devices may remain undetected and unidentified. This would allow an attacker to enter restricted areas of the building, thereby violating the buildings’ security policy.

As SANA’s authors omit some critical details, we analyze SANA under a range of reasonable assumptions. Tamarin finds attacks on SANA for some of these assumptions while proving the protocol to be secure under others, highlighting the importance of these details. We disclosed the issues we found to SANA’s authors. All our models are available at [22] and additional details can be found in the bachelor thesis [25].

A. The SANA Protocol

One central concern of SANA is separating the act of attesting the software state of devices from other administrative duties. The software attestation is carried out by one of potentially many verifying parties. In contrast, the network is administrated by the network owner. To accommodate this separation, SANA is divided into two subprotocols. The *Token Request* subprotocol is an offline authorization protocol, where a verifier obtains a signed authorization token from the network owner. Afterwards, this verifier can use said token to run the *Attestation* subprotocol, which checks whether the devices on the network are running approved software.

More specifically, the devices participating in SANA have one or more of the roles described below. Figure 1 provides an overview of the role assignment of one protocol run.

Verifier \mathcal{V} initiates both the *Token Request* and *Attestation* subprotocols and verifies the attestation result. A verifier possesses an asymmetric key pair $(sk_{\mathcal{V}}, pk_{\mathcal{V}})$. Multiple verifiers may be present in the network, but only one participates in any given protocol run.

Provers \mathcal{P}_i are the devices subject to software attestation. When challenged, they compute an attestation response that is a signed hash over their software. Provers execute all protocol code in a tamper-proof secure execution environment. Thus, we assume that all provers behave honestly in our analysis. Each prover is initialized in a trusted environment, where it obtains an asymmetric key pair (sk_i, pk_i) and an authentic copy of the network owner’s public key $pk_{\mathcal{O}}$.

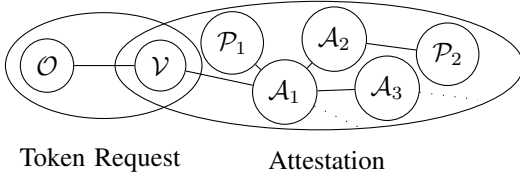


Fig. 1: Overview of SANA's aggregation tree

Network Owner \mathcal{O} initializes the provers. In *Token Request*, the owner authorizes a verifier \mathcal{V} to attest the network. It has knowledge of all devices present in the network and their public keys ($pk_{\mathcal{V}}$ and all pk_i). Owners are assumed to be honest in [12].

Aggregators \mathcal{A}_i distribute \mathcal{V} 's messages to provers and other aggregators and aggregate the provers' attestation responses. This aggregation is presented to \mathcal{V} as the attestation result.

1) *Optimistic Aggregate Signatures*: While aggregate signatures reduce storage and bandwidth requirements, they usually do not reduce the verification time complexity. Thus, the authors of SANA propose *Optimistic Aggregate Signatures* (OAS), a variant of traditional aggregate signatures. As most provers are expected to sign the same message, OAS reduce the verification complexity by specifying a default message M and a separate aggregation of the public key.

More specifically, an OAS scheme is a tuple of algorithms (OAS.gen, OAS.aggPK, OAS.sign, OAS.agg, OAS.vfyAgg), where OAS.gen, OAS.sign, and OAS.agg are defined as for traditional aggregate signatures. OAS.aggPK aggregates multiple public keys into an aggregate public key. OAS.vfyAgg provides more information compared to its traditional counterpart. Given an aggregate signature, an aggregate public key, and a default message M , OAS.vfyAgg either outputs \perp upon failure or otherwise a set \mathcal{B} that for each signed message $m_i \neq M$ contains a tuple (m_i, S_i) , where S_i is the set of public keys whose corresponding secret keys were used to sign m_i . The formal definition of an OAS scheme and its correctness and unforgeability notions are given in Appendix D.

No formal models of OAS schemes exist at the time of writing. However, the proposed OAS scheme provides a similar functionality to traditional aggregate signatures, whereby the main difference lies in an optimized verification algorithm. Since the performance of cryptographic algorithms is usually not accounted for when analyzing protocols in the symbolic model, the described OAS scheme can be modeled with our traditional aggregate signature models.

2) *The Token Request Subprotocol*: *Token Request* is executed between a verifier and the network owner and authorizes the verifier to execute a single instance of the *Attestation* subprotocol. This allows for the attestation to be a public service while mitigating DoS attacks.

A simplified version of *Token Request* is depicted in Figure 2. The signatures exchanged in this subprotocol are generated with a normal digital signature scheme, not the OAS scheme introduced above. To initiate a protocol run, a

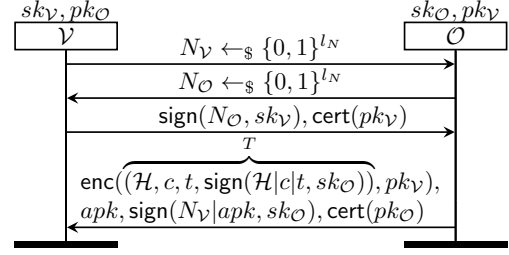


Fig. 2: Simplified SANA *Token Request* subprotocol

verifier \mathcal{V} samples a random nonce $N_{\mathcal{V}}$ and sends it to the network owner \mathcal{O} , which responds with its own nonce $N_{\mathcal{O}}$. Next, \mathcal{V} computes and sends a signature over $N_{\mathcal{O}}$ together with a public key certificate $\text{cert}(pk_{\mathcal{V}})$, issued by a trusted third party, to \mathcal{O} . \mathcal{O} then assembles and signs the authorization token T , which consists of a set \mathcal{H} of hash values of all valid software configurations, a counter value c , which provides replay protection, and an expiry timestamp t . \mathcal{O} aggregates the provers' public keys pk_1, \dots, pk_n into the aggregate public key apk and sends the following back to \mathcal{V} : The token T , encrypted under $pk_{\mathcal{V}}$, apk , a signature over apk and $N_{\mathcal{V}}$, and a public key certificate $\text{cert}(pk_{\mathcal{O}})$. Finally, \mathcal{V} decrypts the token. Naturally, if the verification of any of the exchanged signatures fails, protocol execution is aborted.

3) *The Attestation Subprotocol*: This subprotocol uses the result of the prior *Token Request* as a starting point. It is intended to provide the verifier with guarantees that all provers are in a good state, i.e. their software version matches the latest non-compromised software version.

The *Attestation* subprotocol follows a simple challenge-response mechanism between the verifier and the provers. The verifier assembles an attestation challenge $Ch = \{N, T\}$, consisting of the previously obtained authorization token T and a fresh nonce N . To distribute Ch , the verifier sends it to a gateway aggregator, which forwards it recursively to its adjacent aggregators and provers. This induces a tree-shaped communication path over the network, called an *aggregation tree*, see Figure 1. Before forwarding Ch , an aggregator validates it by verifying the signature and counter in T . The authors claim to mitigate DoS attacks with this mechanism. A prover, upon receiving a challenge, first validates the token contained therein and then computes a hash value h over its software. It then signs the message $m := h|N|c$ using the OAS scheme and returns the signature to its parent in the aggregation tree. Aggregators collect and aggregate all received signatures and send them back towards the root of the aggregation tree. The gateway aggregator presents the aggregation of all responses to the verifier, which then verifies it against the aggregate public key received from the network owner during *Token Request*. Note that the generation of hashes for software attestation is out of SANA's scope.

B. Security Properties

We first present the security properties we considered when analyzing the *Token Request* subprotocol in isolation and then

present the security properties for the combination of both subprotocols.

1) *Token Request Properties*: The authors of SANA [12] do not state any concrete security properties to be satisfied by *Token Request*. They only state properties for the whole attestation scheme, given below. Therefore, we make minimal assumptions and evaluate what properties the protocol satisfies. Specifically, since *Token Request* serves as an authorization mechanism, we analyze the protocol with respect to the increasingly strict notions of entity authentication, introduced by Lowe [23].

2) *Attestation Properties*: We consider two properties.

Attestation Agreement. This is a formalization of what SANA’s authors call *Unforgeability and Freshness*, which they define as follows: If an honest verifier was able to verify an attestation result including a given uncompromised prover, then the claimed integrity measurement reflects the state of the prover’s software during the protocol run. We formalize this property as non-injective agreement between the verifier and any prover, parameterized on the prover’s attestation response.

Token Agreement. This states that a prover should act upon receiving a token only if that token was issued by the owner. We formalize this as non-injective agreement between the owner and a given prover, parameterized on the token.

While the first property is stated as part of the main requirements for SANA in [12], we include the second in our analysis to validate the effectiveness of *Token Request* as an authorization mechanism.

C. Modelling Assumptions

The authors of [12] make the assumption that each device in the network can identify and communicate with its direct neighbors. Moreover, the network’s topology may change dynamically. To simplify our analysis and to account for topology changes, we assume a fully connected network, where each device can communicate with every other network device. We also assume the adversary has full control over the network, allowing the adversary, for example, to delay or drop messages. This is standard in the symbolic model, and ensures that verification results are valid in practice, even in the worst case.

Furthermore, the authors of SANA mandate an existing trusted third-party public key infrastructure (PKI) used by \mathcal{O} and \mathcal{V} , which authenticates the public keys $pk_{\mathcal{O}}$ and $pk_{\mathcal{V}}$ using signed certificates. As the PKI’s capabilities are not further specified, we employ a standard model of a PKI, where the PKI and certificates are abstracted as a public dictionary that binds public keys to identities. As this PKI is independent of SANA, we assume it is unaware of SANA’s roles and thus, it does not bind the parties’ protocol roles to their public key.

As stated in Section VI-A, each prover is initialized with an authentic copy of the owner’s public key. In contrast, the initialization of the *verifiers* is not described by SANA’s authors. Thus, we analyzed the protocol under two different assumptions: First, assuming that verifiers are only initialized

TABLE III: Results for the *Token Request* subprotocol.

Property	Subject	Initialization \mathcal{V}	
		None	Owner Identity
Aliveness	Owner	Proven	Proven
	Verifier	Falsified	Proven
Weak Agreement	Owner	Falsified	Falsified
	Verifier	Falsified	Falsified

with their own secret key $sk_{\mathcal{V}}$ and, second, additionally assuming the initialization with the owner’s identity to look up its public key in the PKI. The first assumption is less restrictive and is based on SANA being motivated as a public attestation service allowing anyone to request an authorization token. Thus, we first assume that verifiers have no prior initialization with the owner. Consultation with the SANA authors, however, revealed the implicit assumption that the owner’s identity is known. This motivates our second assumption about verifiers being initialized with the owner’s identity.

The standard PKI model allows the adversary to compromise individual secret keys. Note that an agent loses most security guarantees when its secret key is compromised. Thus, security properties only consider the guarantees provided to non-compromised, i.e., honest, agents that act in the presence of compromised agents. The authors of SANA also assume the owner to be honest. This assumption is indeed needed, as otherwise Tamarin finds trivial attacks where the adversary uses the owner’s secret key to issue a fake authorization token.

D. Results

We first present the results of analyzing the *Token Request* subprotocol separately and then the results of our analysis of the combination of both subprotocols.

1) *Token Request Analysis*: The results of our analysis of *Token Request* are summarized in Table III, where the subject denotes the party to which the property is guaranteed. We note that only the weakest two notions of entity authentication, Aliveness and Weak Agreement, need to be checked, as Weak Agreement fails to hold, as explained below, and thus all stronger properties also fail.

When assuming that the verifier is *not* initialized with the owner’s identity, Tamarin finds counterexamples for all notions of entity authentication except for Aliveness of the verifier from the perspective of the owner. Assuming verifier initialization with the owner’s identity the protocol additionally provides Aliveness of the owner. We provide the three counterexamples corresponding to the weakest notions of authenticity in Appendix E. Here, we highlight the flaws that enable these attacks.

The *Token Request* subprotocol provides a signing oracle that allows an adversary to obtain a valid signature over an arbitrary term under any verifier’s secret key, provided this verifier initiates an execution of *Token Request*. The adversary can provide any message instead of the nonce $N_{\mathcal{O}}$, which the verifier will then sign and provide to the adversary. This is possible as this term is not cryptographically tied to the rest of the protocol execution. In the counterexample for Aliveness

from the verifier’s perspective, the adversary obtains the apparent owner’s signatures over both the aggregate public key and the token using the signing oracle. Thus, the targeted verifier \mathcal{V} obtains a forged token without the owner’s involvement. Note that this attack relies on the fact that \mathcal{V} accepts another verifier’s public key $pk_{\mathcal{V}}$ as an owner’s public key. Thus, the initialization of the verifier with the owner’s identity prevents this attack.

Tamarin also finds attacks on Weak Agreement from the perspective of both the owner and verifier, for both initialization assumptions. These attacks exploit the fact that the exchanged signatures do not authenticate the designated receiver. Most notably, SANA’s authorization token, which authorizes a specific verifier to perform network attestation, is not bound to that verifier. In the attack from the perspective of the verifier, the targeted verifier receives a token that was intended for another (e.g. the adversary). This attack shows that the adversary can forward a token to an arbitrary verifier, even if the owner would ordinarily refuse to issue a token to the latter.

The counterexample for the Weak Agreement property, from the perspective of the network owner, exploits that the verifier’s signature over the owner’s nonce does not bind the owner identity to this protocol run. The attack results in the owner and verifier disagreeing on their communication partner. Namely, the adversary lets the owner believe that a verifier is communicating with them, while the verifier believes it is communicating with a different owner. This would require that verifiers may attest networks with different owners. It is unclear whether SANA would be deployed in such a scenario and thus whether such an attack could be exploited in practice.

Although the exact exploitation of these attacks is unclear, they clearly show that SANA’s *Token Request* only provides weak authentication properties. Even with the stronger initialization assumptions, the protocol only guarantees Aliveness, i.e., the parties know that their communication partner executed the protocol at some point, but cannot be sure that their communication partner intends to communicate with them in any given protocol run. This rather weak property is achieved by a four message protocol that involves fresh nonces. Note, however, that a simpler two message protocol could achieve the same level of authentication and that slightly modifying the signed messages in the original protocol results in the much stronger Injective Agreement property, as described in [25].

2) *Complete Analysis*: We present the results of our analysis of the combination of both subprotocols in Table IV. We consider four model classes. Each class consists of different models, each using a different aggregate signature model. Table IV specifies the used aggregate signature models, referring to the numbers from Table I.

The first two model classes assume standard adversary capabilities, as described in Section VI-C. As in Section VI-D1, we make two assumptions on the verifier initialization. First, we assume the verifier is only initialized with its own key pair and, second, we assume the verifier is additionally initialized with the owner’s identity. The weakest of our aggregate signature models, Model 4, already results in an attack when

TABLE IV: Results for the complete protocol analysis

		No Keys in apk	Non-Honest	Non-Honest Keys in apk	
		Init \mathcal{V} : None	Init \mathcal{V} : Owner Identity	No Rogue Keys	Rogue Keys
Used Aggregate Signature Models		Model 4	Models 1-6	Models 1, 2, 4, 5	Models 3, 6
Property	Attestation Agreement	Falsified	Proven	Proven	Falsified
	Token Agreement	Proven	Proven	Proven	Proven

assuming no verifier initialization. The stronger assumption of initializing the verifier with the owner identity is verified by Tamarin for each of our six aggregate signature models, see Table I. The second two model classes assume an additional adversary capability, where the adversary can register their non-honest keys with the owner to add them to the aggregate public key apk . While this stronger adversary model is not very realistic for SANA, it showcases the impact of the rogue public key attacks. Namely, Tamarin finds attacks for the two aggregate signature models with the rogue key attack, while Tamarin proves the properties for the other aggregate signature models. Next, we present the results of these four model classes in more detail.

Tamarin proves the Token Agreement property for each of our four model classes. This property is achieved through the authentication of the token by the owner and the fact that every prover is initialized with an authentic copy of the owner’s public key. Note that this *prover* initialization is independent of our assumptions on the *verifier* initialization.

For the model where the verifier is not initialized with the owner’s identity, Tamarin finds counterexamples for the Attestation Agreement property. This attack exploits the signing oracle in the *Token Request* subprotocol described in Section VI-D1 to obtain signatures for both a fake authorization token and one or more fake attestation responses. This attack enables the adversary to present an arbitrary attestation result to the targeted verifier, i.e., it can make the targeted verifier believe that there are no devices with invalid software in the network while the attestation on the potentially compromised provers was never carried out. See Appendix E for more details on this attack.

The above attack relies on the adversary putting a verifier’s public key into the aggregate public key of the forged token. Thus, preventing the adversary from forging authorization tokens also prevents the above attack. This is achieved by assuming the verifier is initialized with the owner’s identity, see Section VI-D1. Accordingly, Tamarin proves Attestation Agreement for the model that take this assumption into account.

Under this additional assumption, Tamarin proves Attestation Agreement for all six aggregate signature models. At first glance it might appear surprising that the insecure and undesirable behaviors of aggregate signatures do not contribute to additional attacks against SANA. However, as these behaviors rely on the presence of non-honest keys, which would need to

be added to the *apk*, they are prevented by the authentication of the aggregate public key *apk*. As the provers are assumed to be initialized by the owner in a secure environment, it is reasonable to assume that the provers' keys are honest. However, to showcase our aggregate signature models, we created models where the adversary can add non-honest keys to the aggregate public key *apk*.

With this additional adversarial capability, the aggregate signature models behave differently. Namely, the two examined properties are proven for the models without rogue public key attacks, Models 1, 2, 4, and 5. Furthermore, Tamarin finds an attack on Attestation Agreement for the models *with* the rogue public key attack, Models 3 and 6. As expected, the adversary can forge an aggregate signature for a targeted prover and a corresponding rogue public key, thus, again forging the attestation result. Note that SANA is specified to have measures against the rogue public key attack implemented. Thus, while showcasing our aggregate signature models, this attack is not realistic. Furthermore, colliding signature attacks and similar undesirable behaviors do not enable attacks against the Attestation Agreement property as the latter requires the considered prover to be honest and have an honest key pair, which is a standard assumption.

This case study of the SANA protocol demonstrates how our different aggregate signature models can be used in practice. It can be sufficient to use Model 4, which supports the fewest behaviors, to find attacks. Models 1-6 can be used to determine necessary assumptions or to prove the security of the protocol. Our case study also highlights the importance of the assumptions on keys: As the insecure and undesirable behaviors of aggregate signatures are based on non-honest keys, some assumptions on keys allow for attacks, while others prevent them.

E. Model Limitations

First, as with all formal methods, the translation of a protocol specification from natural language to a formal model relies on our interpretation of the (informal) specification. We confirmed some of our assumptions with SANA's authors. Our second assumption on the verifier initialization, described in Section VI-C, is based on this exchange.

Second, various parts of the SANA protocol have an iterative and potentially unbounded characteristic, e.g., the aggregation of attestation responses. This resulted in the verification process aborting due to resource exhaustion in Tamarin. As a result, we considered only a small number of provers in our models, namely, two or three (depending on the model). Additional vulnerabilities may arise when considering more or even an unbounded number of provers. The presented attacks for small numbers of provers of course extend to larger numbers. However, the security claims are limited by such bounds.

Finally, we performed several protocol simplifications. For a detailed discussion of all simplifications with arguments for their soundness, see [25]. While our attacks should be applicable in an actual deployment, we could not confirm this, as no implementation of the SANA protocol is publicly available.

VII. CONCLUSION

We developed the first symbolic models of aggregate signatures for Tamarin. We created two classes of models: validation models based on the computational definition of aggregate signatures, and attack-finding models based on Tamarin's built-in signatures. These classes of models are inspired by the signature models by Jackson et al. [15].

Our generic models are based on the general definition of aggregate signatures, and to model BLS aggregate signatures we extended our general aggregate signature models with known attacks on BLS. We extended the attack-finding model for colliding signatures on aggregate signatures and we extended both classes of models for the rogue public key attack.

Our new aggregate signature models enable the analysis of protocols that rely on aggregate signatures, which we demonstrate with the SANA protocol [12]. Our analysis revealed important conditions that must be met during agent initialization to prevent attacks. These conditions were not explicitly stated by the authors of SANA.

The arbitrary number of aggregated signatures, and the loops required to model such aggregations, make Tamarin's termination challenging. Thus, some model simplifications were required for the proofs of some properties to terminate. When using our models in larger protocols, proof techniques, such as oracles and inductive reuse lemmas, tailored to the protocol in question could help with termination. Other options to improve termination could involve dedicated support in Tamarin, for example through improved heuristics.

Our techniques to model arbitrarily many aggregated signatures include the quantification over the elements in a multiset in lemmas and restrictions, using indices in multisets to represent vectors, and using dedicated adversary rules to transform signatures into an aggregation. These techniques are general and could be applied to other multi-party primitives, such as threshold signatures or group signatures.

REFERENCES

- [1] B. Edgington, "Upgrading Ethereum — 2.9.1 BLS Signatures — eth2book.info," https://eth2book.info/capella/part2/building_blocks/signatures/, 2023, [Accessed 27-04-2024].
- [2] Cloudflare, "Cryptography — drand.love," <https://drand.love/docs/cryptography/>, [Accessed 27-04-2024].
- [3] Dfinity, "Chain-key signatures — Internet Computer — internetcomputer.org," <https://internetcomputer.org/how-it-works/threshold-ecdsa-signing/>, [Accessed 27-04-2024].
- [4] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps," in *Advances in Cryptology — EUROCRYPT 2003*, E. Biham, Ed. Berlin, Heidelberg: Springer, 2003, pp. 416–432.
- [5] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International conference on the theory and application of cryptography and information security*. Springer, 2001, pp. 514–532.
- [6] D. Boneh, S. Gorbunov, R. S. Wahby, H. Wee, C. A. Wood, and Z. Zhang, "Draft-irtf-cfrg-bls-signature-05 - BLS Signatures," Jun. 2022. [Online]. Available: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/05/>
- [7] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography - Version 0.5*, 2020. [Online]. Available: <http://toc.cryptobook.us/>

[8] A. Lysyanskaya, “Unique Signatures and Verifiable Random Functions from the DH-DDH Separation,” in *Advances in Cryptology — CRYPTO 2002*, M. Yung, Ed. Berlin, Heidelberg: Springer, 2002, pp. 597–612.

[9] N. T. M. Quan, “0,” Tech. Rep. 323, 2021. [Online]. Available: <https://eprint.iacr.org/2021/323>

[10] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, “ProVerif 2.03: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial,” Sep. 2021. [Online]. Available: <https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf>

[11] B. Schmidt, S. Meier, C. Cremers, and D. Basin, “Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties,” in *2012 IEEE 25th Computer Security Foundations Symposium*. Cambridge, MA, USA: IEEE, Jun. 2012, pp. 78–94. [Online]. Available: <http://ieeexplore.ieee.org/document/6266153/>

[12] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, “Sana: Secure and scalable aggregate network attestation,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 731–742. [Online]. Available: <https://doi.org/10.1145/2976749.2978335>

[13] D. Boneh, S. Gorbunov, R. Wahby, H. Wee, and Z. Zhang, “Draft-irtf-cfrg-bls-signature-04 - BLS Signatures,” [Online]. Available: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/04/>

[14] N. T. M. Quan, “Attacks and weaknesses of BLS aggregate signatures,” Tech. Rep. 377, 2021. [Online]. Available: <https://eprint.iacr.org/2021/377>

[15] D. Jackson, C. Cremers, K. Cohn-Gordon, and R. 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*, ser. CCS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2165–2180. [Online]. Available: <https://doi.org/10.1145/3319535.3339813>

[16] B. Blanchet, “An Efficient Cryptographic Protocol Verifier Based on Prolog Rules,” in *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society, Jun. 2001, pp. 82–96.

[17] C. Cremers, A. Peltonen, and M. Zhao, “An extended hierarchy of security notions for threshold signature schemes and automated analysis of protocols that use them,” Cryptology ePrint Archive, Paper 2024/1920, 2024. [Accessed 15-04-2025]. [Online]. Available: <https://eprint.iacr.org/2024/1920>

[18] J. Le-Papin, B. Dongol, H. Treharne, and S. Wesemeyer, “Verifying List Swarm Attestation Protocols,” in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec ’23. New York, NY, USA: Association for Computing Machinery, Jun. 2023, pp. 163–174. [Online]. Available: <https://dl.acm.org/doi/10.1145/3558482.3581778>

[19] X. Hofmeier, “Formalizing Aggregate Signatures in the Symbolic Model,” Master’s thesis, ETH Zurich, Department of Computer Science, Oct. 2021. [Online]. Available: <https://www.research-collection.ethz.ch/handle/20.500.11850/511820>

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

[21] Various, “Tamarin prover manual,” https://tamarin-prover.com/manual/master/book/001_introduction.html, [Accessed 27-04-2024].

[22] X. Hofmeier, A. Raguso, R. Sasse, D. Jackson, and D. Basin, “One for all: Formally verifying protocols which use aggregate signatures,” 2025, tamarin models of aggregate signatures and the SANA case study. [Online]. Available: <https://doi.org/10.5281/zenodo.15357177>

[23] G. Lowe, “A hierarchy of authentication specifications,” in *Proceedings 10th Computer Security Foundations Workshop*. Rockport, MA, USA: IEEE Comput. Soc. Press, 1997, pp. 31–43. [Online]. Available: <http://ieeexplore.ieee.org/document/596782/>

[24] T. Pornin and J. P. Stern, “Digital Signatures Do Not Guarantee Exclusive Ownership,” in *Applied Cryptography and Network Security*. Berlin, Heidelberg: Springer, 2005, pp. 138–150.

[25] A. Raguso, “Formal Analysis of Network Attestation and Aggregate Signatures,” Bachelor Thesis, ETH Zurich, Department of Computer Science, Mar. 2023. [Online]. Available: <https://www.research-collection.ethz.ch/handle/20.500.11850/731316>

[26] D. Kapur, P. Narendran, and L. Wang, “An E-unification Algorithm for Analyzing Protocols That Use Modular Exponentiation,” in *Rewriting Techniques and Applications*, R. Nieuwenhuis, Ed. Berlin, Heidelberg: Springer, 2003, pp. 165–179.

[27] D. Boneh, B. Lynn, and H. Shacham, “Short Signatures from the Weil Pairing,” in *Advances in Cryptology — ASIACRYPT 2001*, ser. Lecture Notes in Computer Science, C. Boyd, Ed. Berlin, Heidelberg: Springer, 2001, pp. 514–532.

APPENDIX A AGGREGATE SIGNATURE DEFINITIONS

In Section II-A, we described the definition of aggregate signatures, its attack game, and EUF-CMA security definition. Here we provide these definitions in more detail according to Boneh and Shoup [7].

We first provide the definition of an aggregate signature scheme according to Boneh and Shoup [7].

Definition 1 (Aggregate signature scheme [7], page 623). *An aggregate signature scheme $\mathcal{SA} = (\text{gen}, \text{sign}, \text{vfy}, \text{agg}, \text{vfyAgg})$ is a signature scheme with two additional efficient algorithms agg and vfyAgg :*

- *A signature aggregation algorithm $\text{agg}(\mathbf{pk}, \sigma)$ takes as input two equal length vectors, a vector of public keys $\mathbf{pk} = (pk_1, \dots, pk_n)$ and a vector of signatures $\sigma = (\sigma_1, \dots, \sigma_n)$. It outputs an aggregate signature σ_{agg} .*
- *The deterministic aggregate verification algorithm $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk})$ takes as input two equal length vectors, a vector of public keys $\mathbf{pk} = (pk_1, \dots, pk_n)$, a vector of messages $\mathbf{m} = (m_1, \dots, m_n)$, and an aggregate signature σ_{agg} . It outputs either true or false.*

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

$$\Pr[\text{vfyAgg}(\text{agg}(\mathbf{pk}, \sigma), \mathbf{m}, \mathbf{pk}) = \text{true}] = 1$$

Next, we provide the attack game for EUF-CMA of aggregate signatures according to Boneh and Shoup [7].

Definition 2 (Attack game for EUF-CMA of agg. signatures [7], page 626). *For a given aggregate signature scheme \mathcal{SA} with message space \mathcal{M} , and a given adversary \mathcal{A} , the attack game runs as follows:*

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

The adversary wins the game if the following conditions hold:

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

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

Lastly, we provide the EUF-CMA security definition of aggregate signatures according to Boneh and Shoup [7].

Definition 3 (Security of aggregate signatures [7], page 626). *We say that an aggregate signature scheme \mathcal{SA} is secure if for all efficient adversaries \mathcal{A} , the quantity $\text{ASIGadv}[\mathcal{A}, \mathcal{SA}]$ is negligible.*

APPENDIX B BLS AGGREGATE SIGNATURES

Here, we provide the details on BLS aggregate signatures. BLS aggregate signature are constructed from bilinear pairings, also called bilinear maps. A bilinear pairing is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ where $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ are cyclic groups of prime order q and e satisfies bilinearity and non-degenerativity [7]. The central property of bilinear pairings which is used to construct BLS signatures is: for all $\alpha, \beta \in \mathbb{Z}_q$ we have

$$e(g_0^\alpha, g_1^\beta) = e(g_0, g_1)^{\alpha\beta} = e(g_0^\beta, g_1^\alpha) \quad (4)$$

where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators.

A. BLS Signatures

We here present the definition of BLS signatures by Boneh and Shoup [7]:

Definition 4 (the BLS signature scheme [7], 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 $\mathcal{S}_{BLS} = (\text{gen}, \text{sign}, \text{vfy})$, has message space \mathcal{M} and works as follows:

- $\text{gen}()$ (key generation algorithm):
 - secret key: $sk \xleftarrow{R} \mathbb{Z}_q$
 - public key: $pk \leftarrow g_1^{sk} \in \mathbb{G}_1$.
- $\text{sign}(m, sk)$: To sign a message $m \in \mathcal{M}$ using a secret key $sk \in \mathbb{Z}_q$, do $\sigma \leftarrow H(m)^{sk} \in \mathbb{G}_0$
- $\text{vfy}(\sigma, m, pk)$: 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) assumption [5]. Boneh et al. [4] define this property as follows:

Definition 5 (co-CDH [4]). *Given $g_a, g_a^x \in \mathbb{G}_a$ and $h \in \mathbb{G}_b$ compute $h^x \in \mathbb{G}_b$.*

Theorem 1 ([7], 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 [5] and Section 15.5.1 of [7].

B. BLS Aggregate Signatures

BLS signatures support signature aggregation. We will derive the aggregation function by first presenting a naive approach, that is not secure, following the presentation in Boneh and Shoup [7].

Definition 6 (Naive BLS aggregate signature [7], page 624). *The aggregate signature scheme $\mathcal{SA}_{BLS} = (\mathcal{S}_{BLS}, \text{agg}, \text{vfyAgg})$:*

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

$$e(\sigma_{agg}, g_1) = e(H(m_1), pk_1) \cdots e(H(m_n), pk_n) \quad (5)$$

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

$$e(\sigma_{agg}, g_1) \stackrel{?}{=} e(H(m), pk_{agg}) \quad (6)$$

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

Note that BLS aggregate signatures cannot be modeled directly in Tamarin using Diffie-Hellman group elements and bilinear pairings, as BLS signature aggregation uses the group operation directly which is not available by the built-in theories that only support exponentiation, due to the undecidability of unification of such equational theories [26].

C. Rogue Public Key Attack

As described on a high level in Section IV-A, the above naive construction is insecure since it is vulnerable to a rogue public key attack. Here, we provide more details on the rogue public key attack.

The adversary creates the rogue public key pk_{rogue} for a target public key pk_{target} by choosing a random value $\alpha \xleftarrow{R} \mathbb{Z}_q$ and computing:

$$pk_{\text{rogue}} \leftarrow g_1^\alpha / pk_{\text{target}} \in \mathbb{G}_1 \quad (7)$$

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 signature aggregation:

$$\sigma_{\text{agg}_{\text{rogue}}} := H(m)^\alpha \in \mathbb{G}_0 \quad (8)$$

This aggregate signature is valid for the message m repeated twice, the target public key pk_{target} and the rogue public key pk_{rogue} :

$\text{vfyAgg}(\sigma_{\text{agg}_{\text{rogue}}}, (m, m), (pk_{\text{target}}, pk_{\text{rogue}}))$ as:

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

The adversary creates a forgery and thus violates the EUF-CMA security definition of aggregate signatures, Definition 3. Thus, this naive BLS aggregate signature scheme needs to be extended with mitigations to prevent the rogue public key attack.

D. Mitigating the Rogue Public Key Attack

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

1) *Prevent Duplicate Messages*: Note that the rogue public key aggregate $\sigma_{\text{agg}_{\text{rogue}}}$ validates for the message m repeated twice. In fact, this attack is only possible with multiple times the same message. The first mitigation, presented by Boneh et al. [4], 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.

2) *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 7 (BLS agg. signature with message augmentation [7], page 626). *Our modified aggregation scheme, denoted $\mathcal{SA}_{BLS}^{(1)}$, is the same as \mathcal{SA}_{BLS} in 6 except that the signing algorithm now uses a hash function $H : \mathbb{G}_1 \times \mathcal{M} \rightarrow \mathbb{G}_0$ and is defined as*

$$\text{sign}(m, sk) := H(pk, m)^\alpha \text{ where } sk = \alpha \in \mathbb{Z}_q \text{ and } pk \rightarrow g_1^\alpha \quad (9)$$

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

$$\begin{aligned} & \text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m} \in \mathcal{M}^n, \mathbf{pk} \in \mathbb{G}_1^n) = \text{true} \\ & \text{if } e(\sigma_{\text{agg}}, g_1) = e(H(pk_1, m_1), pk_1) \cdots e(H(pk_n, m_n), pk_n) \end{aligned} \quad (10)$$

As described in Appendix B-B, the naive approach for BLS aggregate signatures of Definition 6 can be verified faster, if all messages are the same. This optimization is not possible with this method.

3) *Proof of Possession of the Secret Key*: This method preserves the fast verification described in Appendix B-B. 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.

Definition 8 (BLS aggregate signature with Proof of Possession [7], page 627). *The modified aggregation scheme, denoted $\mathcal{SA}_{BLS}^{(2)}$, is the same as \mathcal{SA}_{BLS} defined in 6 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 $H' : \mathbb{G}_1 \rightarrow \mathbb{G}_0$, and operate as follows:

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

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 $\pi_1, \dots, \pi_n \in \mathbb{G}_0$ 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 $\mathcal{SA}_{BLS}^{(1)}$ and $\mathcal{SA}_{BLS}^{(2)}$ are secure in the sense of Definition 3, assuming co-CDH holds for e , and the hash functions H and H' are modeled as random oracles, see [4] and [7] Section 15.5.3.3 for the proofs.

E. Attacks on BLS Signatures

In Section IV-B1, we described the splitting zero attack by Quan [9] on a high level. Here we provide more details on this attack. We first describe the attack on single BLS signatures and then the extension to BLS aggregate signatures.

1) *Identity Element Keys with a Single BLS Signature*: Quan [9] first presents an attack on single BLS signatures. The adversary chooses the secret key to be equal to zero, this results in the corresponding public key and all signatures signed with it being the identity element:

$$sk_{\text{adv}} := 0 \quad (11)$$

$$pk_{\text{adv}} := g_1^{sk_{\text{adv}}} = g_1^0 = 1 \quad (12)$$

$$\sigma_{\text{adv}} := H(m)^{sk_{\text{adv}}} = H(m)^0 = 1 \quad (13)$$

As $\text{vfy}(\sigma_{\text{adv}}, m, pk_{\text{adv}}) = \text{true}$ 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 $m \in \mathcal{M}$.

The BLS IETF draft [13] requires in the verification algorithm to check that the public key is not the identity element. According to Quan [9], some libraries implementing the IETF draft omit this check or implement it incorrectly, which makes this attack practical for those implementations.

2) *Splitting Zero Attack*: As described in Section IV-B1, Quan [9] extends this attack for BLS aggregate signatures and bypasses the identity element check by choosing two non-honest keys, such that $sk_{\text{mal}_1} + sk_{\text{mal}_2} = 0$. And both sk_{mal_1} and sk_{mal_2} are non-zero. The adversary then signs some message

m with each non-honest 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 non-honest public keys is also the identity element:

$$pk_{\text{mal}_1} \cdot pk_{\text{mal}_2} = 1 \quad (14)$$

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

$$\sigma_{\text{agg}_{1,2,3}} := \sigma_{\text{mal}_1} \cdot \sigma_{\text{mal}_2} \cdot \sigma_3 = \sigma_3 \quad (15)$$

This aggregate signature will validate against the message 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

$\text{vfyAgg}(\sigma_{\text{agg}_{1,2,3}}, (m', m', m_3), (pk_{\text{mal}_1}, pk_{\text{mal}_2}, pk_3)) = \text{true}$ 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 mentioned in Section IV-B1, this attack does not violate the EUF-CMA security definition of aggregate signatures, Definition 3. The attack violates uniqueness, which is not guaranteed by EUF-CMA security. This might be surprising, as single BLS signatures are unique and thus popular in blockchain schemes. Thus, it is important to note that BLS aggregate signatures are not unique and thus do not provide consensus. Other primitives or protocols have to provide consensus, this cannot be achieved by aggregate signatures which only provide EUF-CMA security.

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. However, the Proof of Possession method does not resolve this problem, as the adversary is in possession of the two non-honest secret keys.

As each subset of public keys with the same signed message could be colluded, the attack could only be prevented by

checking not only all pairs, but all subsets of public keys. For a large number of signatures with the same signed message, this would be quite expensive.

APPENDIX C

VALIDATION MODEL RESTRICTIONS IN TAMARIN

In Section III-A, we provided the restrictions that define the validation models and in Section III-B, we mentioned some aspects on how we formalized these restrictions in Tamarin. Here we provide more details on this formalization in Tamarin. We discuss the correctness restriction as an example. We reformulate Restriction 1 on Page 5 to the following equivalent restriction:

Restriction 4.

$$\begin{aligned}&\forall \mathbf{pk}, \mathbf{m}, \sigma. \text{vfyAgg}(\text{agg}(\sigma), \mathbf{m}, \mathbf{pk}, \text{false}) \\ &\Rightarrow (\exists i \in \{1, \dots, n\}. (\neg \sigma_i = \text{sign}(m_i, sk_i) \vee \neg \text{Honest}(pk_i)))\end{aligned}$$

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 VfyAgg(agggregation, mAndPk, false)@i
4 ==>
5 ((Ex si ind thetaAgg mi ski thetaMPk.
6   VfyAgg(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   VfyAgg(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 $\text{vfyAgg}(\sigma_{\text{agg}}, \mathbf{m}, \mathbf{pk}, \text{false})$ with the action fact $\text{VfyAgg}(\text{agggregation}, \text{mAndPk}, \text{false})$. 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 VfyAgg 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 handle the case of two or more aggregated signatures and lines 10 to 13 handle the case of one aggregated signature. To explain why we need this distinction, we first explain how we express the existential quantification of the right-hand side. The existential quantification $\exists i \in \{1, \dots, n\}$ quantifies over the signatures, messages, and secret keys with the same index. We now look at how we express $\exists \sigma_i \in \sigma$. The aggregation $\text{agg}(\sigma)$ is represented as

$$\text{agg}(\langle s1, \text{ind1} \rangle + \dots + \langle sn, \text{indn} \rangle)$$

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

$$\text{agg}(\langle si, \text{ind} \rangle + \text{thetaAgg})$$

See the following equation:

$$\langle \sigma_1, i_1 \rangle + \langle \sigma_2, i_2 \rangle + \dots + \langle \sigma_n, i_n \rangle \equiv_{AC} \langle \sigma_i, i_i \rangle + \underbrace{\langle \sigma_1, i_1 \rangle + \dots + \langle \sigma_{i-1}, i_{i-1} \rangle + \langle \sigma_{i+1}, i_{i+1} \rangle + \dots + \langle \sigma_n, i_n \rangle}_{\vartheta_{agg}}$$

Thus we can express the existential quantification $\exists \sigma_i \in \sigma$ 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 `VfyAgg` in lines 5 and 6 with `aggregation` replaced with `agg(<si, ind>+thetaAgg)`. We discuss these different formulations at the end of this section.

Now back to the case distinction: In our models, 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 added a second restriction to cover the case of one aggregated signature.

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

We noted above, that we can express $\sigma_i \in \sigma$ 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:

```
1 restriction
   OneMessageKeyPairPerSignature_RestateActionFact:
2   "All si thetaAgg ind messagesKeys #i.
3   VfyAgg(agg(<si, ind>+thetaAgg), messagesKeys,
4   true)@i
5   ==> Ex mi ski thetaMPk.
6   VfyAgg(agg(<si, ind>+thetaAgg)
7   , <mi, pk(ski), ind> + thetaMPk, true)@i"
8 restriction OneMessageKeyPairPerSignature_Equation:
9   "All si thetaAgg ind messagesKeys #i.
10  VfyAgg(agg(<si, ind>+thetaAgg), messagesKeys,
11  true)@i
12  ==> Ex mi ski thetaMPk.
13  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 `VfyAgg` after the implication, while the second explicitly equates `<mi, pk(ski), ind>+thetaMPk` and `messagesKeys`. 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.

APPENDIX D OAS DEFINITIONS

In Section VI-A1 we provided an overview of the *Optimistic Aggregate Signature* (OAS) scheme by [12]. Here we provide their definitions.

Definition 9 (OAS, Adapted from [12]). *An Optimistic Aggregate Signature (OAS) scheme is a quintuple of probabilistic polynomial time algorithms (OAS.gen, OAS.aggPK, OAS.sign, OAS.agg, OAS.vfyAgg). Let $l \in \mathbb{N}$ be the security parameter.*

- $\text{OAS.gen}(1^l) \rightarrow (pk, sk)$ generates a secret signing key sk and a public verification key pk .
- $\text{OAS.aggPK}(pk_1, \dots, pk_n) \rightarrow apk$ aggregates public keys pk_1, \dots, pk_n into an aggregate public key apk .
- $\text{OAS.sign}(m, M, sk) \rightarrow \sigma$ generates a signature σ given a message $m \in \{0, 1\}^*$, a default message $M \in \{0, 1\}^*$ and a secret key sk .
- $\text{OAS.agg}(\sigma_1, \sigma_2, M) \rightarrow \sigma_{agg}$ aggregates the aggregate signatures σ_1 and σ_2 into the aggregate signature σ_{agg} .
- OAS.vfyAgg is the verification algorithm. Given a set of non-contributing signers S_\perp , an aggregate signature σ_{agg} , a default message M and an aggregate public key apk it outputs either \perp if the signature is invalid or a set $\mathcal{B} = \{(m_i, S_i) | i \in \{1, \dots, \mu\}\}$, where S_i is the set of public keys pk_i whose corresponding signing key sk_i was used to sign the message m_i .

Definition 10 (Correctness of an OAS, adapted from [12]). *An OAS scheme is correct if*

- for all $l \in \mathbb{N}$, all public $(pk, sk) \leftarrow \text{OAS.gen}(1^l)$, all S_\perp with $pk \notin S_\perp$ and all $m, M \in \{0, 1\}^*$ we have $\Pr[\text{OAS.vfyAgg}(S_\perp, \text{OAS.sign}(m, M, sk), M, \text{OAS.aggPK}(S_\perp \cup \{pk\})) = \mathcal{R}] = 1$, where $\mathcal{R} = \emptyset$ if $m = M$ and $\mathcal{R} = \{m, \{pk\}\}$ if $m \neq M$.
- aggregation works. Meaning that for all valid aggregate signatures σ_1, σ_2 , all distinct sets S_1, S_2 , all subsets $S_{1,\perp} \subseteq S_1, S_{2,\perp} \subseteq S_2, M \in \{0, 1\}^*$, if $\text{OAS.vfyAgg}(S_{1,\perp}, \sigma_1, M, \text{OAS.aggPK}(S_1)) \rightarrow B_1$ and $\text{OAS.vfyAgg}(S_{2,\perp}, \sigma_2, M, \text{OAS.aggPK}(S_2)) \rightarrow B_2$, then $\Pr[\text{OAS.vfyAgg}(S_{1,\perp} \cup S_{2,\perp}, \text{OAS.agg}(\sigma_1, \sigma_2), M, \text{OAS.aggPK}(S_1 \cup S_2)) = B_1 \sqcup B_2] = 1$, where $B_1 \sqcup B_2$ denotes the set union except that if there exist m^*, S, S' s.t. $(m^*, S) \in B_1$ and $(m^*, S') \in B_2$, then $B_1 \sqcup B_2$ contains the element $(m^*, S \cup S')$ instead of the two original tuples (m^*, S) and (m^*, S') .

Informally, OAS unforgeability mandates that no adversary can generate an aggregate signature that attributes a message to an honest signer which never signed said message, even if all other signers are compromised.

Definition 11 (Unforgeability of OAS, adapted from [12]). *Let $l \in \mathbb{N}$. Let OAS be an OAS scheme, \mathcal{A} an adversary and Q the set of messages queried by \mathcal{A} to its signing oracle $\text{OAS.sign}(\cdot, sk)$. We define the unforgeability game for OAS*

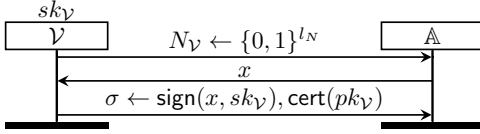


Fig. 3: The signing oracle $\text{SO}(\mathcal{V}, x)$ provides a signature over an arbitrary term x under the secret key of any verifier \mathcal{V} .

as follows:

```

(pk, sk) ← OAS.gen( $1^l$ )
( $\sigma, S_\perp, (pk_1, \dots, pk_n), (sk_1, \dots, sk_n)$ ) ←  $\mathcal{A}^{\text{OAS.sign}(\cdot, sk)}(pk)$ 
If  $\exists i. pk_i \neq pk \wedge (pk_i, sk_i) \notin \text{OAS.gen}(1^l)$  then return 0
 $S \leftarrow \{pk_1, \dots, pk_n\}$ 
 $apk \leftarrow \text{OAS.aggPK}(S)$ 
 $B \leftarrow \text{OAS.vfyAgg}(S_\perp, \sigma, M, apk)$ 
If  $S_\perp \not\subseteq S \vee \exists (m_i, S_i) \in B. S_i \not\subseteq S$  then return 0
If  $\exists (m_i, S_i) \in B. pk \in S_i \wedge m_i \notin Q$  then return 1
 $S_M \leftarrow S \setminus (S_\perp \cup \bigcup_{(m_i, S_i) \in B} S_i)$ 
If  $pk \in S_M \wedge M \notin Q$  then return 1 else return 0

```

OAS is unforgeable if for all polynomial-time adversaries \mathcal{A} , the probability that the unforgeability game outputs 1 is negligible.

APPENDIX E ATTACKS ON SANA

In this section, we provide the message sequence charts and some details on the attacks targeting SANA, described in Section VI-D. We first present the attacks targeting only the *Token Request* subprotocol, and second we present attacks targeting the complete SANA protocol.

A. Attacks on Token Request

We provide the three counterexamples corresponding to the weakest notions of authenticity the protocol fails to satisfy, namely, Aliveness from the verifier's perspective where the verifier is initialized only with its secret key, and Weak Agreement from the perspective of the verifier and owner which applies to both initialization scenarios.

The attack on Aliveness from the perspective of the verifier relies on the *Token Request*'s signing oracle, provided in Figure 3. We introduce the notation $\text{SO}(\mathcal{V}, x)$ to represent an invocation of this oracle to obtain a signature over the term x under \mathcal{V} 's secret key $sk_{\mathcal{V}}$.

We provide the counterexample for Aliveness from the verifier's perspective where the verifier is initialized with only its secret key in Figure 4. The attack is described in Section VI-D. This attack forms the basis of our attacks on the complete protocol, assuming the verifier is *not* initialized with the owner identity.

We provide the attack on the Weak Agreement property from the perspective of the verifier in Figure 5. After a verifier initiates a protocol run with the owner, all further messages are intercepted by the adversary, which first finishes the execution with the owner and, thus, obtains a token encrypted under the

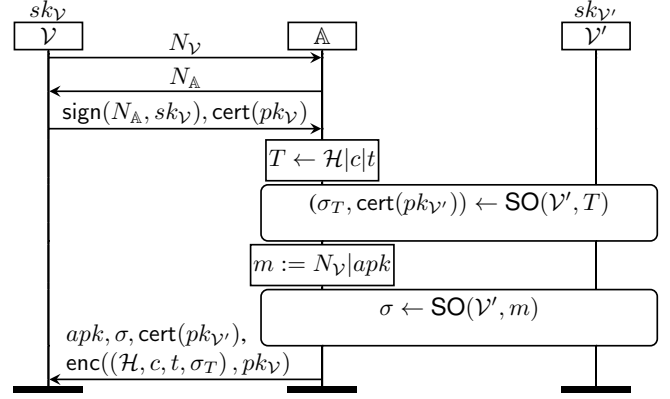


Fig. 4: Counterexample for the Aliveness property from the perspective of the verifier \mathcal{V} .

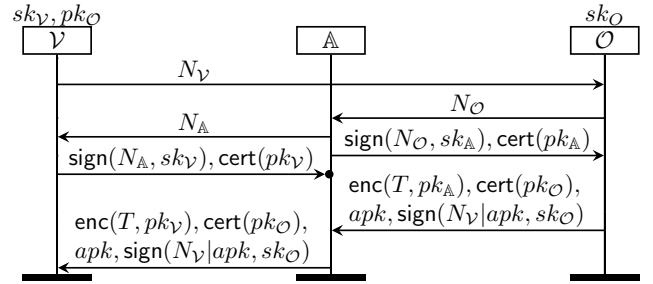


Fig. 5: Counterexample for the Weak Agreement property from the perspective of the verifier \mathcal{V} .

adversary's public key and a signature generated under the owner's secret key. The adversary can then re-encrypt said token under the verifier's public key and send it back to the verifier together with the owner's signature. Therefore, the verifier must believe that it communicated with the owner, while the latter actually issued a token to the adversary.

We provide the counterexample for Weak Agreement from the network owner's perspective in Figure 6. In this attack, the verifier \mathcal{V} starts the *Token Request* with owner \mathcal{O}_2 while the adversary forwards the verifier's messages to owner \mathcal{O}_1 . Thus, the verifier \mathcal{V} and the owner \mathcal{O}_1 do not agree on their communication partner which contradicts Weak Agreement.

B. Attacks on the Complete Protocol

In this section, we provide the attacks on the complete protocol targeting Attestation Agreement. First, we present the attack based on the signing oracle, assuming the verifier is initialized only with its secret key. Second, we present the attack assuming the verifier is initialized with the owner's identity but assuming the adversary can add rogue public keys to the aggregate public key.

The first attack is depicted in Figure 7. The adversary exploits the signing oracle in the *Token Request* subprotocol (see Figure 3) to obtain signatures for both a fake authorization token and one or more fake attestation responses. Note that neither the network owner nor any of the provers need to be active at any time. The attack is solely executed between the

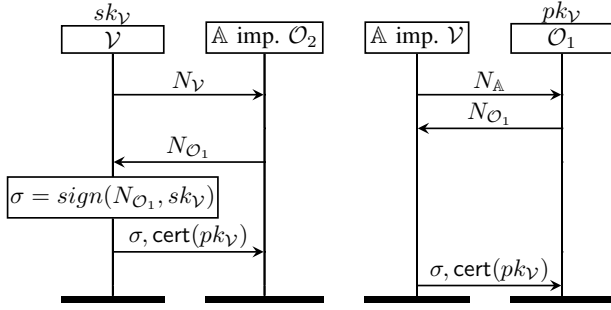


Fig. 6: Counterexample for the Weak Agreement property from the perspective of the network owner \mathcal{O}_1 . \mathbb{A} impersonating \mathcal{X} is shortened to \mathbb{A} imp. \mathcal{X}

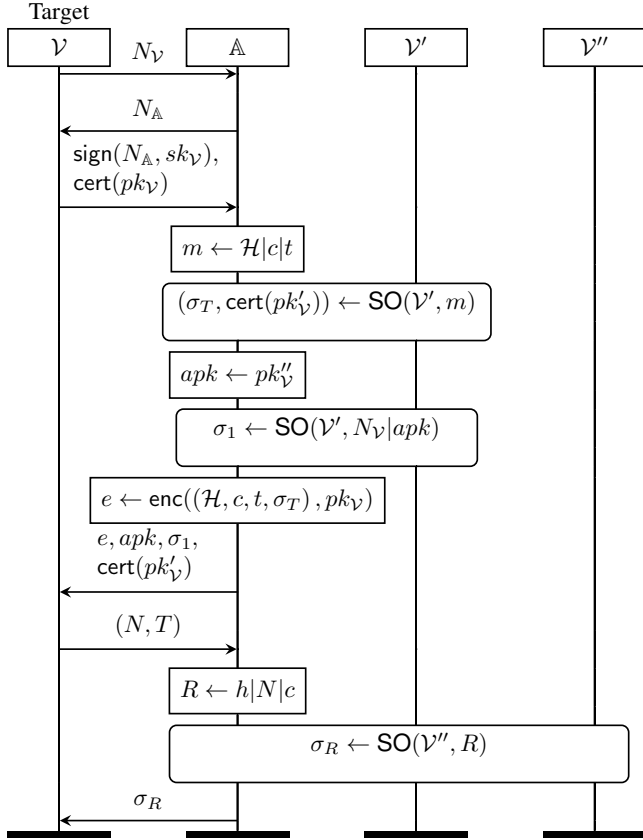


Fig. 7: Counterexample for the Attestation Agreement property, assuming no verifier initialization.

adversary, the target verifier, and a few supporting verifiers as signing oracles.

This attack enables the adversary to present an arbitrary attestation result to the targeted verifier, i.e., it can make the targeted verifier believe that there are no devices with invalid software in the network while the attestation on the potentially compromised provers was never carried out. This distorts not only the verifier's perception of the devices' state but also of what devices even exist. The aggregate public key should inform the verifier of the list of provers in the network. However, as the adversary can forge this, it can provide the

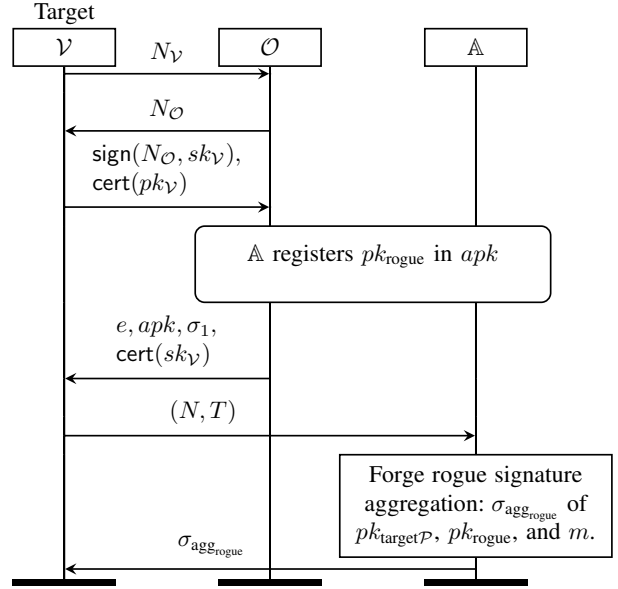


Fig. 8: Counterexample for the Attestation Agreement property, assuming rogue public key registration.

verifier with other verifier's public keys instead of the prover's public keys.

Regarding the practicality of this attack, we highlight two considerations. First, each invocation of the signing oracle requires the corresponding verifier to initiate a protocol run. However, some signatures can be obtained offline as the corresponding messages are known beforehand. Also, potential mechanisms causing a verifier to re-run the protocol upon failure may be exploited for repeated oracle access. Second, the attack relies on the assumption that the signatures from *Token Request* can be aggregated in the OAS scheme. This is possible if BLS signatures [27] are used,² as outlined in Appendix F. Since SANA targets IoT devices, which often require small code bases, it is likely that developers choose BLS signatures with an OAS extension instead of two independent signature libraries, making this assumption realistic.

The attack, assuming the adversary capability of registering rogue public keys in the aggregate public key, is depicted in Figure 8. The *Token Request* is performed normally, while the attestation response is forged by the adversary. The adversary registers a rogue public key pk_{rogue} for a target prover's public key pk_{targetP} and provides the rogue signature aggregation $\sigma_{\text{agg_rogue}}$ of these keys and message m to the verifier. This forges the target prover's attestation response.

APPENDIX F AGGREGATING BLS SIGNATURES AND OAS

In this section, we show how a valid BLS signature can be transformed into a valid OAS signature, which can then be further aggregated into an aggregate OAS signature. We begin by repeating the concrete OAS scheme given in [12].

²The use of the OAS scheme in *Token Request* is explicitly excluded by [12].

Definition 12 (Concrete OAS scheme construction according to [12]). Let $\mathbb{G}_0, \mathbb{G}_1$ and \mathbb{G}_T be multiplicative groups of prime order p with generators g_0, g_1 and g_t and let $e : \mathbb{G}_0 \times \mathbb{G}_1 \rightarrow \mathbb{G}_T$ be an efficiently computable bilinear map. Let $H : \{0, 1\}^* \rightarrow \mathbb{G}_1$ be a hash function.

Key Generation Secret keys are sampled uniformly random: $sk \leftarrow_{\$} \mathbb{Z}_p$. Then, $pk \leftarrow g_1^{sk}$.

Public Key Aggregation Aggregating public keys pk_1, \dots, pk_n is done via multiplication: $apk = \prod_{i=1}^n pk_i$.

Signing Let $m \in \{0, 1\}^*$. Then, $\tau \leftarrow H(m)^{sk}$. If m is the default message, then $\sigma = \{\tau, \emptyset\}$. Otherwise, $\sigma = \{\tau, \{(m, \{pk\})\}\}$.

Signature Aggregation Let $\sigma_1 = \{\tau_1, \mathcal{B}_1\}$ and $\sigma_2 = \{\tau_2, \mathcal{B}_2\}$. Then, σ_1 and σ_2 are aggregated as follows: $\sigma \leftarrow \{\tau_1 \cdot \tau_2, \mathcal{B}_1 \sqcup \mathcal{B}_2\}$, where \sqcup is defined as in Definition 10, Appendix D.

Verification Let S_{\perp} be the set of non-contributing public keys, apk the aggregate public key, M the default message and $\sigma = (\tau, \{(m_1, S_1), \dots, (m_{\mu}, S_{\mu})\})$ the aggregate signature. Let,

$$apk_M \leftarrow apk \cdot \left(\prod_{pk \in S_{\perp}} pk \cdot \prod_{i=1}^{\mu} \prod_{pk \in S_i} pk \right)^{-1}$$

The verification algorithm returns true if

$$e(\tau, g_1) = e(H(M), apk_M) \cdot \prod_{i=1}^{\mu} e \left(H(m_i), \prod_{pk \in S_i} pk \right)$$

and false otherwise.

Let OAS denote the OAS scheme described above. Let BLS denote the BLS signature scheme with the same groups $\mathbb{G}_0, \mathbb{G}_1$ and \mathbb{G}_T , the same bilinear map e and the same hash function H . Let $\sigma_{BLS} = H(m)^{sk}$ be a BLS signature over the message m generated with the key pair (sk, pk) using the algorithms of BLS. We define $\sigma_{OAS} := (\sigma_{BLS}, \mathcal{B})$, where $\mathcal{B} = \emptyset$ if $m = M$ or $\mathcal{B} = \{(m, \{pk\})\}$ otherwise. We now argue that $\text{OAS.vfyAgg}(\emptyset, \sigma_{OAS}, M, pk) = \text{true}$. To see this, consider two cases.

If $m = M$, then $apk_M = pk$ and, thus,

$$\begin{aligned} e(\sigma_{BLS}, g_1) &\stackrel{(1)}{=} e(H(m), pk) \\ &= e(H(M), apk_M). \end{aligned}$$

If $m \neq M$, then $apk_M = 1$ and, therefore,

$$\begin{aligned} e(\sigma_{BLS}, g_1) &\stackrel{(1)}{=} e(H(m), pk) \cdot 1 \\ &= e(H(m), pk) \cdot e(H(M), apk_M). \end{aligned}$$

In (1) we use the correctness of BLS.