Forward-Anonymity for Anix: Formalization and Symbolic Analysis of a Blackout-Resistant Anonymous Broadcast System

by

Yuheng Wen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2026

© Yuheng Wen 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Anix [11] is a blackout-resistant anonymous microblogging protocol supporting remote trust establishment and message endorsement. However, compromise of a user's current public identity key may link earlier pseudonyms to that user. To address this limitation, this thesis proposes a one-way key-ratcheting mechanism and uses it to augment Anix. We also give the first formalism of a forward-anonymous message broadcasting system, named FAMB, of which the augmented Anix protocol is an instantiation. We define FAMB through algorithms, protocol state, adversarial oracles, and security games, and prove correctness and one-wayness of the ratcheting construction in the random-oracle model. We then build symbolic models of the Anix-based instantiation in Tamarin and automatically prove soundness under both the standard signature abstraction and the weaker SVS model. Attempts to verify unlinkability expose an issue in Tamarin's observational-equivalence mode and practical scalability limitations. Finally, we review related Tamarin unlinkability proofs and identify common simplification patterns.

Acknowledgements

I would first like to express my sincere gratitude to my supervisor, Professor Samuel Jaques. Academically, I have benefited greatly from his originality, rigorous approach to research, and remarkable breadth of knowledge. His ability to discover unexpected connections and develop elegant ideas continually encouraged me to think more deeply, while his insistence on precision taught me how proofs should be constructed, examined, and communicated. Beyond academia, the integrity, fairness, and consideration with which he approaches people and responsibilities showed me the importance of conducting oneself with principle and professionalism. I will carry both of these lessons with me well beyond this thesis.

I am also grateful to the professors and students in the Department of Combinatorics and Optimization, especially those in the cryptography community, for creating such a friendly and supportive environment. I thank my friends and my partner for their encouragement and support throughout this journey. Finally, I owe my deepest gratitude to my parents for their unconditional love, patience, and support.

Table of Contents

Author's Declaration **ii**

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

1.1 Background 1

1.2 Contribution 2

2 Preliminaries **4**

2.1 Tamarin-Prover 4

2.1.1 Security Model 4

2.1.2 Symbolic Model 4

2.1.3 Under the Hood 5

2.1.4 Basics 6

2.1.5 Diff Mode 8

2.2 Random Oracle Model 8

2.3 Blinded Signature Scheme 9

2.4 Public Key Encryption 15

2.5 Anix 16

2.5.1 Background 16

2.5.2 Anix Protocol 17

3 Forward-Anonymous Anix: FAMB **20**
3.1 Protocol Description 20
3.1.1 Security Game 21
3.2 Instantiation 31

4 Key Ratcheting **37**
4.1 Forward Anonymity 37
4.1.1 Key-Ratcheting Scheme 38
4.1.2 Security Game 39
4.1.3 Construction 40
4.1.4 Proof of Correctness 40
4.1.5 Proof of Security 40

5 Proving FAMB in Tamarin **47**
5.1 Modeling Abstraction 47
5.2 soundness_std.spthy 48
5.2.1 Cryptographic primitives (L11–L13) 48
5.2.2 Restrictions (L17–L24) 49
5.2.3 Key generation and user creation (L26–L37) 50
5.2.4 Ratchet (L39–L44) 51
5.2.5 Adversarial Key Ratcheting (L46–L66) 51
5.2.6 Trust establishment (L68–L75) 52
5.2.7 Ticking (advancing time) (L77–L98) 53
5.2.8 Corruption (L100–L112) 54
5.2.9 Signing and verification (L119–L135) 55
5.2.10 The unforgeability lemma (L145–L151) 55
5.2.11 Executability lemmas (L155–L196) 56
5.2.12 Proofs Interpretation 57

5.3 soundness.spthy 61
5.4 Towards Proving Unlinkability 64
5.4.1 Model 1 64
5.4.2 Model 2 66
5.4.3 Issue in Diff mode 67

6 Unlinkability in Other Models **68**
6.1 5G-AKA Untraceability 68
6.2 ECC-DAA Unlinkability 70
6.3 Others 71
6.4 Discussion 71

7 Conclusion **73**

References **75**

Chapter 1

Introduction

1.1 Background

Internet shutdowns are increasingly used to restrict the flow of information during periods of political unrest. When Internet access is unavailable, mobile mesh networks allow nearby devices to exchange and relay messages using technologies such as Bluetooth and Wi-Fi Direct, without relying on centralized communication infrastructure. Such systems are particularly valuable for citizens and activists who need to distribute information and coordinate quickly. In this setting, a useful communication system should support one-to-many broadcasting, protect the anonymity of message authors, and help users determine whether received messages and their sources are trustworthy. Prior empirical work confirms that these demands arise in real protest environments. Albrecht et al. found protesters sought pseudonymity and forward anonymity [2].

Anix was proposed to address these requirements. It provides anonymous microblogging over a mesh network, allows users to establish trust remotely, and supports anonymous message endorsement so that users can assess messages based on the opinions of their trusted contacts [11]. Its prototype and simulation results also suggest that these mechanisms can be implemented efficiently and remain useful in blackout scenarios. Anix therefore provides an attractive basis for studying anonymous and trustworthy communication in a decentralized setting.

Anix achieves selective recognizability through pseudonyms and blinded signatures. A pseudonym cannot normally be linked to its author, but a user who knows the author's public identity key can derive the corresponding blinded verification key and recognize

pseudonyms generated under that identity. The public identity key is therefore not globally published; it is selectively shared through Anix's trust-establishment mechanism. This allows a user to control which other users can recognize their messages while remaining anonymous to the rest of the network.

However, this selective linkability also creates the security issue that motivates this thesis. The same public identity key may be used to recognize multiple pseudonyms until that identity is replaced or revoked. If the key is later exposed through device compromise or coercion, an adversary can test previously observed pseudonyms against it and link the corresponding past messages to the user. Identity revocation can prevent the compromised identity from being used safely in the future, but it does not by itself make pseudonyms already generated under that identity unlinkable. In other words, Anix does not directly provide the stronger notion of forward anonymity, under which compromising a user's current public identity key should not reveal which earlier messages were sent by that user.

To address this problem, this thesis augments Anix with a one-way key-ratcheting mechanism that evolves both a user's identity key pair and the public identity keys stored by followers. The resulting protocol, called FAMB, is formalized through explicit algorithms and security games.

1.2 Contribution

This thesis makes the following contributions:

- We propose a key-ratcheting component for Anix to provide forward anonymity. We give a concrete construction and prove its correctness and one-wayness in the random-oracle model.
- We give the first formalization of Anix with the proposed key-ratcheting component. We define the resulting protocol as a forward-anonymous message broadcast system, called FAMB, and specify its algorithms, protocol state, adversarial oracles, security games, and concrete instantiation.
- We develop a symbolic model of FAMB in Tamarin and prove its soundness using both the standard signature model and a relaxed signature model (SVS), expressed as an unforgeability property, under the modeling assumptions stated in this thesis.

- During the unlinkability analysis, we identify and report an issue in Tamarin's observational-equivalence mode involving the interaction between adversary-controlled inputs and restrictions.
- We review the existing Tamarin examples that prove related unlinkability and untraceability properties. From these examples, we summarize common modeling patterns and give recommendations for using Tamarin's diff mode on complex, stateful protocols.

Chapter 2

Preliminaries

2.1 Tamarin-Prover

This chapter introduces Tamarin Prover, which is the verification tool used in the analysis chapter of this thesis. Tamarin is an automated prover for the symbolic analysis of security protocols, developed by Basin, Cremers, Dreier, Sasse, and contributors [12] in 2013.

2.1.1 Security Model

Tamarin assumes an active network adversary, also known as the Dolev-Yao adversary [7]. In the Dolev-Yao model, the adversary has full control of the network. He can intercept, read, modify, and drop any message in transit. He can also decompose messages into their parts following honest primitives. For example, if the message is in the form of an encryption, the adversary requires private keys to be able to decompose the message (actually ciphertext) received in the channel to its corresponding message. The adversary can also construct new messages from known terms by applying public functions and sending these messages to users.

2.1.2 Symbolic Model

Tamarin is based on the so-called symbolic model [13, 1] . Symbolic means that we abstract away bitstrings to algebraic terms. Specifically, messages are no longer bitstrings, but terms. In contrast to the symbolic model is the computational model, where messages

are modeled as bitstrings, which enables one to reason about details such as key length or message length. In addition, the symbolic model treats primitives as perfect black boxes, whereas the computational model treats them as functions on bit strings. The attacker in the computational model can be any probabilistic polynomial-time Turing machine, whereas in the symbolic model, we focus on the knowledge that can be deduced by the adversary.

2.1.3 Under the Hood

Protocols and adversaries in Tamarin are specified using an expressive language based on multiset rewriting rules. These rules compose a labeled transition system in which the state of this system records the adversary's knowledge, the messages on the network, the fresh values that have been generated, and the current state of the protocol. The transitions correspond to rule applications through which the adversary and the protocol consume and produce messages. Security goals can then be expressed in two ways: as trace properties checked against the traces of the transition system, or as observational equivalence between two transition systems, which is how Tamarin captures indistinguishability properties.

To prove a lemma, Tamarin first negates the security property and adds it to a constraint system that already encodes the protocol model. It then performs a backward search over this system, repeatedly applying reduction steps that refine a partial dependency graph. A typical step picks an unresolved goal, such as a fact whose origin in the protocol rules must be identified, an action required by the negated formula, or a disjunction in the constraints, and replaces it with the cases under which the goal can be satisfied. Each branch of the search closes in one of two ways: its constraints become impossible so no execution can satisfy them, or no further reduction applies and the remaining constraints already describe a complete execution. If every branch closes with impossible constraints, the lemma is proved, since it means a counterexample of the security property is proven impossible. If some branch instead terminates with an execution, that execution is a dependency graph witnessing an attack.

Note that since security protocols can be infinite state and there can be unbounded number of sessions of the protocol, the correctness of security protocols is an undecidable problem, Tamarin may not terminate on a given verification problem. Hence, users may need to use Tamarin's interactive mode to explore the proof states, inspect attack graphs, and manually guide the proof search to make it terminate.

2.1.4 Basics

A Tamarin theory is a text file in the `.spthy` format. It declares the function symbols and equations that extend the built-in algebra, defines the protocol as a set of multiset-rewriting rules, lists the security lemmas, and optionally adds restrictions and helper lemmas. We now walk through some basic concepts in Tamarin in more detail.

Terms, sorts and facts. Messages are modeled as *terms* in an order-sorted term algebra [15]. *Sorts* are sets of terms. There is one top sort *msg* and two subsort *pub* and *fresh*. *pub* (denoted by text strings enclosed in single quotes, e.g. 'string') is for public constants, which represent known or guessable constants such as identities. *Fresh* is for fresh values (written with a leading $\sim$) such as freshly generated random values, keys, or nonces, which we consider to be hidden from the adversary. Facts come in two flavors. *Linear* facts, written *F(...)*, are 'consumed' on a match. We explain what this means in the next paragraph. *Persistent* facts, written *!F(...)* with a leading exclamation mark, are never 'consumed' and live throughout the traces. The reserved facts *Fr*, *In*, *Out* and K have fixed meaning: freshness, network input, network output, adversary knowledge.

Rules. The basic unit is a rule of the shape

$$[\ L\ ]\ \texttt{--}\ [\ A\ ] \rightarrow [\ R\ ]$$

where L is a multiset of *premise* facts, A is a multiset of *action* facts written on the trace, and R is a multiset of *conclusion* facts. At any point during an execution, Tamarin implicitly maintains a current protocol state, also called a configuration. This state is a multiset of facts that have been produced by earlier rule applications and are available to be used by later rules. In order for a rule to fire, the premise facts in L must be matched against facts in this current state. Matching means that Tamarin finds facts in the current state with the required shape. Once the rule fires, the matched linear premise facts are consumed, meaning that they are removed from the current state. Matched persistent facts are not removed and remain available for future rule applications. After the premise fact is consumed, the conclusion fact is produced and added to the current protocol state. The action multiset acts like a tag for the rule. It labels that the specific rule is fired at a timestamp. It is invisible to the protocol state but is what security lemmas reason over.

Lemmas. A lemma is a security claim written as a first-order logic formula over protocol traces. Note that lemmas reason only over action facts and not linear facts. The protocol

state itself is invisible to them. A typical secrecy property is written

```
All n k #i.  Sent(n,k)@#i ==> not (Ex #j.  K(n)@#j)          (2.1)
```

where `K` is the reserved adversary-knowledge fact. This translates to: "For all n, k, and timestamp i, if there is an action fact Sent(n,k) at timestamp i, this must imply there does not exist a timestamp j such that K(n) appears at timestamp j". It means that for every `n` marked `Sent` under `k`, the adversary never derives `n`. By default, a lemma is implicitly `all-traces` and must hold on every execution. The keyword `exists-trace` switches the interpretation to existential and requires only a single witnessing trace, typically used in an `executable` sanity-check lemma. If the proof does not terminate, the user can intervene with annotations such as `[reuse]` or `[use_induction]`, an auxiliary sources lemma, or a goal-ranking oracle.

Sources lemmas. Some lemmas, in particular those whose proof would otherwise loop on partial deconstructions, need a helper called a `[sources]` lemma. The annotation tells Tamarin where a specific term must have originated, so that the prover can rule out spurious sources during search. Source lemmas need to be proven first, and often proved using induction.

Restrictions. Restrictions are written exactly like lemmas but are treated as assumptions rather than goals. They restrict the set of traces to be considered when exploring traces.

Proving. Tamarin runs proofs in two modes. The batch mode

```
tamarin-prover file.spthy --prove
```

attempts every lemma with the default heuristic; the interactive mode

```
tamarin-prover interactive file.spthy
```

opens a browser GUI in which proofs can be inspected and steered by hand. Proof debugging at this level usually happens in the interactive mode.

2.1.5 Diff Mode

The lemmas of the previous subsection are trace properties: they ask whether any single trace of the protocol violates a formula. Several properties of interest cannot be phrased this way. Vote privacy, anonymity, unlinkability and key-indistinguishability all have the shape "the adversary cannot tell world A from world B", and that is a statement about a pair of systems, not about any one trace.

The `diff` operator. Inside a rule, the user writes `diff(t1, t2)`, which means "use t_1 on the left, t_2 on the right". Tamarin generates two instances of every rule that contains a `diff` term: one for the left world and one for the right. Rules without `diff` are shared between the two worlds unchanged.

Bi-systems and mirroring. The combined model is a *bi-system*. Observational equivalence holds when, for every reachable behaviour on one side, the adversary can also realise the corresponding behaviour on the other. In proof terms, every rule firing in the left world carries a *mirroring obligation* for the right world (and vice versa), discharged by an extra proof method MIRRORED. A mirroring failure, that is, a step that fires in one world but cannot be mirrored on the other, plays the same role as a falsified trace lemma: it is the candidate attack on indistinguishability. Note that observational equivalence uses approximations.

2.2 Random Oracle Model

The random oracle model is a popular heuristic for hash functions that is used to justify cryptographic constructions. The idea is simply to model a hash function $\mathcal{H}$ as a function $\mathcal{O}$ that is chosen at random from all possible functions. When we do security analysis, we can replace $\mathcal{H}$ with $\mathcal{O}$. In addition, in the random oracle model, we also assume that the adversary can obtain the value of $\mathcal{O}$ at an arbitrary input. So we treat $\mathcal{O}$ as a random oracle, and we say security properties "fall into the random oracle model" if we prove security using this random oracle.

2.3 Blinded Signature Scheme

A digital signature scheme is a tuple of three algorithms $(\mathit{KeyGen}, \mathit{Sign}, \mathit{Verify})$. The algorithm KeyGen generates a signing key and a public verification key pair (sk, pk). Given a signing key sk and a message m, the signing algorithm Sign creates a signature σ. The verification algorithm Verify takes as input a public key pk, a message m, and a signature σ, and outputs 1 if the signature is valid, and 0 otherwise.

A signature scheme with key blinding [9, 6] contains an additional blinding step before the key is used. It extends the syntax of a digital signature scheme with the algorithms $(\mathit{BlGen}, \mathit{BlPubKey}, \mathit{BlSign}, \mathit{Verify})$. The algorithm BlGen generates a blinding key bk. The algorithm $\mathit{BlPubKey}$ takes as input a public key pk, a blinding key bk, and a context ctx, and outputs a blinded public key bpk. The algorithm BlSign takes as input a signing key sk, a blinding key bk, a message m, and a context ctx, and produces a signature σ. Verify takes in a blinded public key bpk and a message and signature pair. Correctness for Verify requires that, for all messages $m \in \{0,1\}^*$ and all contexts $ctx \in \{0,1\}^*$, it holds that:

$$\Pr\left[\mathsf{Verify}(bpk, m, \sigma) = 1 \;\middle|\; \begin{array}{l} (sk, pk) \leftarrow \mathsf{KeyGen}() \\ bk \leftarrow \mathsf{BlGen}() \\ bpk \leftarrow \mathsf{BlPubKey}(pk, bk, ctx) \\ \sigma \leftarrow \mathsf{BlSign}(sk, bk, m, ctx) \end{array}\right] = 1.$$

Eaton et al. proposed two security definitions for signature scheme with key blinding. The first one is unlinkability. Informally, it means that an adversary without knowledge of the long-term public key who only observes blinded public keys and their corresponding signatures cannot distinguish between a blinding of the long-term public key and a blinding of a freshly generate public key. It is formally specified in Algorithm 1.

Algorithm 1: $\mathsf{Unlinkability}^{\mathcal{A}}(\lambda)$

$\Sigma \leftarrow \emptyset$;
$(sk_S, pk_S) \leftarrow \mathsf{KeyGen}()$;
$s \leftarrow 0$;
$b \leftarrow \{0, 1\}$;
$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{BlPublicKey}}, \mathcal{O}_{\mathsf{Challenge}}, \mathcal{O}_{\mathsf{BlKeySign}}}()$;
return $b = b'$;

Oracle $\mathcal{O}_{\mathsf{BlPublicKey}}()$: **begin**
  $bk \leftarrow \mathsf{BlGen}()$;
  $bpk \leftarrow \mathsf{BlPubKey}(pk_S, bk)$;
  $\Sigma \leftarrow \Sigma \cup \{(bpk, bk)\}$;
  return bpk;
end

Oracle $\mathcal{O}_{\mathsf{Challenge}}()$: **begin**
  if $s = 1$ **then**
    return $\perp$;
  end
  $s \leftarrow 1$;
  if $b = 0$ **then**
    $(sk'_S, pk'_S) \leftarrow \mathsf{KeyGen}()$;
    $sk^*_S \leftarrow sk'_S$;
    $pk^*_S \leftarrow pk'_S$;
  end
  else
    $sk^*_S \leftarrow sk_S$;
    $pk^*_S \leftarrow pk_S$;
  end
  $bk^* \leftarrow \mathsf{BlGen}()$;
  $bpk^* \leftarrow \mathsf{BlPubKey}(pk^*_S, bk^*)$;
  $\Sigma \leftarrow \Sigma \cup \{(bpk^*, bk^*)\}$;
  return bpk^*;
end

Oracle $\mathcal{O}_{\mathsf{BlKeySign}}(m, bpk)$: **begin**
  if $s = 1$ ***and*** $bpk = bpk^*$ **then**
    return $\mathsf{BlSign}(sk^*_S, bk^*, m)$;
  end
  if $(bpk, bk) \notin \Sigma$ **then**
    return $\perp$;
  end
  return $\mathsf{BlSign}(sk_S, bk, m)$;
end

The second security definition is unforgeability. It is defined mostly the same as it is for classical digital signatures, except that the signing requests and forgeries are made with respect to the blinded public keys. Strong unforgeability is formally defined in Algorithm 2.

Algorithm 2: $\mathsf{Unforgeability}^{\mathcal{A}}(\lambda)$

$\Sigma \leftarrow \emptyset$;
$(sk, pk) \leftarrow \mathsf{KeyGen}()$;
$(m^*, bk^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{BlKeySign}}}(pk)$;
if $bk^* = \perp$ **then**
  $pk^* \leftarrow pk$;
end
else
  $pk^* \leftarrow \mathsf{BlPubKey}(pk, bk^*)$;
end
if $\mathsf{Verify}(pk^*, m^*, \sigma^*) = 1$ **then**
  if $(bk^*, m^*, \sigma^*) \notin \Sigma$ **then**
    return True;
  end
end
return False;

Oracle $\mathcal{O}_{\mathsf{BlKeySign}}(m, bk)$: **begin**
  if $bk = \perp$ **then**
    $\sigma \leftarrow \mathsf{Sign}(sk, m)$;
  end
  else
    $\sigma \leftarrow \mathsf{BlSign}(sk, bk, m)$;
  end
  $\Sigma \leftarrow \Sigma \cup \{(bk, m, \sigma)\}$;
  return σ;
end

Intuitively, key blinding adds one step before the key is used, in which a hash function derives a blinding factor to transform the original key pair into a new blinded key pair used for signing and verification. The constructions are specified in Algorithm 3 and Algorithm 4.

Algorithm 3: EdDSA with Key Blinding

Algorithm $\mathsf{KeyGen}()$: **begin**
  $sk \leftarrow \{0,1\}^{256}$;
  $(\perp, pk, \perp) \leftarrow \mathsf{InnerKeyGen}(sk)$;
  return (sk, pk);
end

Algorithm $\mathsf{InnerKeyGen}(sk)$: **begin**
  $h \leftarrow \mathsf{SHA512}(sk)$;
  $x \leftarrow \mathsf{ScalarClamp}(h[0:32])$;
  $A \leftarrow xG$;
  $pre \leftarrow h[32:64]$;
  return (x, A, pre);
end

Algorithm $\mathsf{BlGen}()$: **begin**
  $bk \leftarrow \{0,1\}^{256}$;
  return bk;
end

Algorithm $\mathsf{BlPubKey}(pk, bk)$: **begin**
  $h \leftarrow \mathsf{SHA512}(bk)$;
  $\beta \leftarrow h[0:32]$;
  $bpk \leftarrow \beta \cdot pk$;
  return bpk;
end

Algorithm $\mathsf{BlSign}(sk, bk, m)$: **begin**
  $(x, A, pre) \leftarrow \mathsf{InnerKeyGen}(sk)$;
  $(\beta, pre') \leftarrow \mathsf{SHA512}(bk)$;
  $r \leftarrow \mathsf{SHA512}((pre \parallel pre'), m)$;
  $R \leftarrow rG$;
  $A' \leftarrow \beta \cdot A$;
  $k \leftarrow \mathsf{SHA512}(R, A', m)$;
  $s \leftarrow (r + kx\beta) \bmod L$;
  return (R, s);
end

Algorithm $\mathsf{Verify}(pk, m, (R, s))$: **begin**
  $A \leftarrow pk$;
  $k \leftarrow \mathsf{SHA512}(R, A, m)$;
  return $sG = R + kA$;
end

In EdDSA with key blinding, KeyGen generates the Ed25519 key pair. In this scheme, the secret key sk is a random 256-bit seed, and the public key pk is derived from it by running InnerKeyGen. The signing algorithm Sign takes the secret key and a message m, and outputs a signature. The verification algorithm Verify takes a public key, a message, and a signature, and checks whether the signature is valid. The value L denotes the order of the Ed25519 group:

$$L = 2^{252} + 27742317777372353535851937790883648493.$$

The value G denotes the fixed generator of the group. Thus, when the algorithm writes $A = xG$, it means that the public key A is obtained by scalar multiplication of the generator G by the secret scalar x. The function ScalarClamp is the standard Ed25519 clamping function. It takes the first 32 bytes of the hash of the secret seed, interprets them as a little-endian integer, and modifies certain bits so that the resulting scalar has the required form for Ed25519 signing.

For EdDSA with key blinding, it is defined over an elliptic curve E of prime order q, with generator G. The secret key is a random scalar

$$sk \in \mathbb{Z}_q^*,$$

and the corresponding public key is

$$pk = sk \cdot G \in E.$$

The function $\overline{C}$ is a conversion function that maps an elliptic curve point to an element of $\mathbb{Z}_q$. The function H2S is a hash-to-scalar function, which maps a blinding key to an element of $\mathbb{Z}_q^*$. The main idea of key blinding is to derive a scalar blinding factor β from the blinding key bk. If $bk = \perp$, then the scheme sets $\beta = 1$, so no blinding is applied. Otherwise, it sets

$$\beta \leftarrow \mathsf{H2S}(bk).$$

The blinded public key is then computed as

$$bpk = \beta \cdot pk.$$

The signing algorithm uses the same factor β in the signature equation, so the resulting signature verifies under the blinded public key bpk rather than the original public key pk.

Based on these definitions, Eaton et al. prove that both EdDSA with key blinding and ECDSA with key blinding satisfy unlinkability and unforgeability under their corresponding assumptions [9].

Algorithm 4: ECDSA with Key Blinding

Algorithm KeyGen(): **begin**
  $sk \leftarrow \mathbb{Z}_q^*$;
  $pk \leftarrow sk \cdot G$;
  return (sk, pk);
end

Algorithm BlGen(): **begin**
  $bk \leftarrow \{0,1\}^{256}$;
  return bk;
end

Algorithm InnerBlind(bk): **begin**
  if $bk = \perp$ **then**
    $\beta \leftarrow 1$;
  end
  else
    $\beta \leftarrow \mathsf{H2S}(bk) \in \mathbb{Z}_q^*$;
  end
  return β;
end

Algorithm BlPubKey(pk, bk): **begin**
  $\beta \leftarrow \mathsf{InnerBlind}(bk)$;
  return $\beta \cdot pk$;
end

Algorithm BlSign(sk, bk, m): **begin**
  $h \leftarrow \mathsf{Hash}(m) \in \mathbb{Z}_q$;
  $\beta \leftarrow \mathsf{InnerBlind}(bk)$;
  $r \xleftarrow{\$} \mathbb{Z}_q^*$;
  $\mathcal{R} \leftarrow r \cdot G$;
  $t \leftarrow \overline{C}(\mathcal{R}) \in \mathbb{Z}_q$;
  if $t = 0 \ \vee \ h + t\beta sk = 0$ **then**
    return fail;
  end
  $s \leftarrow r^{-1}(h + t\beta sk)$;
  return (s, t);
end

Algorithm Verify($bpk, m, (s, t)$): **begin**
  $h \leftarrow \mathsf{Hash}(m) \in \mathbb{Z}_q$;
  $\mathcal{R} \leftarrow s^{-1}h \cdot G + s^{-1}t \cdot bpk$;
  if $\mathcal{R} \neq \mathcal{O} \ \wedge \ \overline{C}(\mathcal{R}) = t$ **then**
    return true;
  end
  else
    return false;
  end
end

2.4 Public Key Encryption

A public-key encryption scheme is a tuple of three efficient algorithms

$$\mathcal{E} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}).$$

The algorithm KeyGen is a probabilistic key-generation algorithm. It outputs a public key and secret key pair

$$(pk, sk) \leftarrow \mathsf{KeyGen}().$$

The encryption algorithm Enc is also probabilistic. It takes as input a public key pk and a message m, and outputs a ciphertext

$$c \leftarrow \mathsf{Enc}(pk, m).$$

The decryption algorithm Dec is deterministic. It takes as input a secret key sk and a ciphertext c, and outputs either a message m or a special reject symbol $\perp$:

$$m \leftarrow \mathsf{Dec}(sk, c).$$

Correctness requires that decryption undoes encryption. More precisely, for all messages m in the message space $\mathcal{M}$, it should hold that

$$\Pr\left[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m \mid (pk, sk) \leftarrow \mathsf{KeyGen}()\right] = 1.$$

Note that the scheme is defined over a message space $\mathcal{M}$ and a ciphertext space $\mathcal{C}$.

The standard security property for public-key encryption is semantic security, or indistinguishability of encryptions. Informally, it says that an adversary who sees an encryption of one of two chosen messages should not be able to tell which message was encrypted. The game is specified in Algorithm 5.

Algorithm 5: $\mathsf{SemanticSecurity}_b^{\mathcal{A}}(\lambda)$

$(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$;
$(m_0, m_1) \leftarrow \mathcal{A}_1(pk)$;
if $m_0, m_1 \notin \mathcal{M}$ **then**
  return $\perp$;
end
if $|m_0| \neq |m_1|$ **then**
  return $\perp$;
end
$b \leftarrow \{0, 1\}$;
$c \leftarrow \mathsf{Enc}(pk, m_b)$;
$b' \leftarrow \mathcal{A}_2(c)$;
return b';

In this game, the bit b determines which of the two messages is encrypted. In experiment 0, the challenger encrypts m_0, while in experiment 1, it encrypts m_1. The adversary wins if it can distinguish these two experiments. Formally, the distinguishing advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\mathsf{sem}}_{\mathcal{E},\mathcal{A}}(\lambda) = \left|\Pr[\mathsf{SemanticSecurity}_0^{\mathcal{A}}(\lambda) = 1] - \Pr[\mathsf{SemanticSecurity}_1^{\mathcal{A}}(\lambda) = 1]\right|.$$

and we say a public-key encryption scheme is semantically secure if this advantage is negligible for every probabilistic polynomial-time adversary $\mathcal{A}$.

2.5 Anix

2.5.1 Background

Anix is a blackout-resistant communication protocol that lets users communicate via a mesh network when out of the Internet. In general, it allows users to anonymously broadcast and endorse messages. It also enables remote trust establishment between users, allowing them to decide whom to trust based on the messages they observe. As a result, trusted users can identify which messages originate from users who trust them. This scheme was proposed by Kamali and Barradas in 2025 [11]. This thesis embarks with the goal of giving a formal proof of the anonymity property using Tamarin.

2.5.2 Anix Protocol

Anix consists of three components: pseudonym and trust management, trust establishment, and message endorsement. In this section, we will describe each component in detail.

Keys

Each user has two types of randomly generated asymmetric keys: *identity keys* (Pk_{ID}, Sk_{ID}) and *one-time-use keys* (Pk_{OTU}, Sk_{OTU}). Each of the key pairs includes separate keys for encryption and signing.

Pseudonyms and Identity Management

Whenever a user broadcasts a message to the mesh network, the message appears to originate from a pseudonym. A pseudonym is generated as follows:

1. Generate a fresh one-time-use encryption key pair (PkE_{OTU}, SkE_{OTU}) and a fresh one-time signing key pair (PkS_{OTU}, SkS_{OTU}).
2. Use blinded key-signing with the user's long-term signing keys (PkS_{ID}, SkS_{ID}) to obtain a blinded signature σ over $PkE_{OTU}||PkS_{OTU}$.
3. Construct the pseudonym as $PSU = PkE_{OTU}||PkS_{OTU}||\sigma$.

More specifically, the signature is generated as specified in Algorithm 6. Note that by this

Algorithm 6: Blinded signing for pseudonym generation

$m \leftarrow PkE_{OTU} \parallel PkS_{OTU}$;
$bf \leftarrow \mathsf{Hash}(m \parallel PkS_{ID})$;
$\widetilde{SkS}_{ID} \leftarrow \mathsf{Hash}(bf, SkS_{ID}) \cdot SkS_{ID}$;
$\sigma \leftarrow \mathsf{Sign}(m, \widetilde{SkS}_{ID})$;
return σ;

design, a pseudonym is only recognizable to someone who knows the signer's long-term public identity key PkS_{ID}, since this key is needed to derive the corresponding blinded verification key for the signature inside the pseudonym. Anix uses this property for trust management.

Trust Management

Anix supports both one-way and bi-directional relationships between users. To avoid ambiguity, we describe this relation using the terms *follow* and *follower*. Suppose Alice sees a message from Bob's pseudonym and finds the content reliable. Alice may then choose to follow Bob. This does not mean that Alice learns Bob's long-term identity. Instead, it means that Alice is willing to reveal her own public identity key to Bob, so that Bob can recognize future pseudonyms generated by Alice.

Thus, the relation is asymmetric. If Alice follows Bob, then Bob learns enough information to recognize Alice's future pseudonyms, but Alice does not learn Bob's identity key. If Bob also follows Alice, then the same procedure is run in the opposite direction, and the relationship becomes bi-directional.

Alice following Bob works as follows:

1. Alice downloads the one-time-use public encryption key of Bob from Bob's pseudonym.
2. Alice encrypts PubK_{ID_A} under Bob's one-time-use public encryption key:
$$\mathsf{OWTMessage} \leftarrow \mathsf{Enc}_{\mathsf{PkE}_{OTU_B}}(\mathsf{PubK}_{ID_A}).$$
3. Alice sends $\mathsf{OWTMessage}$ to Bob through the mesh network.
4. Bob decrypts the received message using the corresponding one-time-use secret encryption key:
$$\mathsf{PubK}_{ID_A} \leftarrow \mathsf{Dec}_{\mathsf{SkE}_{OTU_B}}(\mathsf{OWTMessage}).$$
5. Bob stores PubK_{ID_A} in his follower list $L_{\mathsf{follower},B}$. From this point on, Bob can verify whether a future pseudonym was generated by Alice.
6. Alice stores Bob's pseudonym or public information in her follow list $L_{\mathsf{follow},A}$. This records that Alice has chosen to follow Bob.

Message Endorsing

The last component is message endorsing. It helps users to reason about the message's trustability in the form of votes, which can be downvotes or upvotes. For instance, if Alice receives a message and the message content seems misleading to her, she can downvote that message to let whoever has her identity key know about her opinion, and pass this

message down the mesh network. In practice, the vote is simply composed by a blinded signature of the hash of the message's full data, concatenated with a single bit 0 or 1 representing downvote or upvote. The blinded signing scheme is the same as used in pseudonym generation. So only users who have Alice's public identity key are able to verify the signature and learn that this vote is made by Alice.

Chapter 3

Forward-Anonymous Anix: FAMB

3.1 Protocol Description

The previous chapter introduced Anix as an anonymous broadcast protocol based on pseudonyms, blinded signatures, and follow relationships. However, the original construction does not address what happens when a user's public identity key is revealed at a later time. Since this key is used to recognize pseudonyms, its exposure may allow previous pseudonyms to be linked to the user. To address this limitation, we augment Anix with a one-way key-ratcheting mechanism. We then introduce FAMB as a formalism for forward-anonymous message broadcasting and describe the augmented Anix protocol as an instantiation of this formalism. In this chapter, ratcheting is treated abstractly as a key-evolution mechanism that updates identity keys over time. The concrete ratcheting construction and its security argument are given in the next chapter.

Adding the key-ratcheting component to Anix, we now formally describe its protocol. When defining the oracles, endorsements are omitted. This is because endorsements use the same blinded-signature mechanism as pseudonym generation. Thus, for the soundness and unlinkability games, it is enough to let the oracles reason about messages and signatures. We give more justification in paragraphs 5.1.

Definition 1. *A* forward-anonymous message broadcast *(FAMB) system is a collection of algorithms with the following syntax and functionality:*

- $\mathsf{FAMB.KeyGen}() \to (SK, PK)$. *This algorithm generates a fresh secret key SK and its corresponding public identity key PK.*

- FAMB.Sign$(SK, m, t) \rightarrow \sigma$. *This algorithm uses the secret key at protocol time t to generate a signature σ on the message m.*
- FAMB.Verify$(PK, m, \sigma) \rightarrow \{0, 1\}$. *This algorithm checks whether σ is a valid signature on m under the public identity key PK.*
- FAMB.TRatchet$(PK, t) \rightarrow PK'$. *This algorithm advances a public identity key by t ratchet steps.*
- FAMB.SRatchet$(PK, SK, t) \rightarrow (PK', SK')$. *This algorithm advances an identity key pair by t ratchet steps while preserving the correspondence between its public and secret keys.*
- FAMB.PseuGen$(P_i, t) \rightarrow (P_i', psu)$. *This algorithm generates a fresh pseudonym psu at protocol time t and returns the updated local state P_i' containing its associated one-time-use keys.*
- FAMB.Follow$(P_i, psu) \rightarrow (P_i', m')$. *This algorithm creates an encrypted follow request using the one-time-use public encryption key contained in psu and updates the sender's follow state.*
- FAMB.AllowFollow$(P_i, m) \rightarrow (P_i', m')$. *This algorithm processes a received follow request, stores the follower's public identity key when decryption succeeds, and returns the updated local state.44*

3.1.1 Security Game

The security game begins with an initialization phase Game.Init(). In this phase, the challenger sets up the public parameters, initializes the protocol state, and samples the randomness needed.

The adversary is then executed with oracle access: $\mathcal{O}_{init}$, $\mathcal{O}_{sign}$, $\mathcal{O}_{verify}$, $\mathcal{O}_{trust}$, $\mathcal{O}_{corrupt}$, $\mathcal{O}_{challenge}$, $\mathcal{O}_{pseugen}$ and $\mathcal{O}_{time}$. These oracles describe the interface between the adversary and the game. They allow the adversary to initialize protocol instances, request signatures, verify signatures, establish trust between users, corrupt users, issue challenge queries, and update the time-dependent parts of the protocol.

In the anonymity game, the adversary outputs a guessing bit b', and wins if $b = b'$. In the soundness and correctness games, the adversary wins if it outputs win, which represents that it has found a violation of the corresponding property. The game is specified as follows:

1. $Game.Init()$
2. $\mathcal{A}^{\mathcal{O}_{init},\mathcal{O}_{sign},\mathcal{O}_{verify},\mathcal{O}_{trust},\mathcal{O}_{corrupt},\mathcal{O}_{challenge},\mathcal{O}_{time},\mathcal{O}_{pseugen}}$
3. - anonymity game wins if $b = b'$
   - soundness game wins if output 'win'
   - correctness game wins if output 'win'

The security games maintain several global data structures. The set U_{honest} contains the users that are currently honest, while $U_{corrupt}$ contains the users that have been corrupted by the adversary. The set S_{valid} records the signatures that were honestly generated during the game. And G_{trust} represents the trust relation between users. An edge $i \to j$ in G_{trust} means that user i trusts user j.

Game.Init. The initialization algorithm sets up the initial state of the game. At the beginning of the experiment, there are no honest users, no corrupted users, no honestly generated signatures, and no trust relations. Hence, U_{honest}, $U_{corrupt}$, S_{valid}, and G_{trust} are all initialized as empty sets. These sets are updated by the oracles during the rest of the security game. The game also maintains an append-only pseudonym board $\mathcal{B}_{psu}$ and a private metadata table $\mathcal{M}_{psu}$. Each published pseudonym is assigned a fresh public handle h. The board entry $\mathcal{B}_{psu}[h]$ contains the pseudonym, while $\mathcal{M}_{psu}[h] = (j, t)$ records its creator and publication time. The metadata table is used only for bookkeeping by the challenger and is not revealed to the adversary.

Algorithm 7: Game.Init

$U_{honest} \leftarrow \emptyset$;
$U_{corrupt} \leftarrow \emptyset$;
$S_{valid} \leftarrow \emptyset$;
$G_{trust} \leftarrow \emptyset$;
$\mathcal{B}_{psu} \leftarrow \emptyset$;
$\mathcal{M}_{psu} \leftarrow \emptyset$;

The verification oracle $\mathcal{O}_{verify}$. The oracle $\mathcal{O}_{verify}$ models verification queries made by the adversary. On input a message m, a signature σ, signature expiry time t abstracted from pseudonym corresponding to the signature, and a verifier identity i, the oracle first

runs the real verification algorithm $FAMB.Verify$ and stores the result in b. It then runs the validity checking oracle $ShouldVerify$ and stores the result in b'. The value b represents the actual output of the cryptographic verification algorithm, while b' represents whether the signature should be accepted according to the history of the protocol execution.

The oracle then compares these two values. If the real verification algorithm accepts the signature, but the ideal predicate says that the signature should not be accepted, then the game aborts with a soundness failure. This means that the adversary has found a signature that is accepted by the scheme even though it should not be valid. Conversely, if the real verification algorithm rejects the signature, but the ideal predicate says that it should be accepted, then the game aborts with a correctness failure. This means that an honestly generated and valid signature was rejected.

In the anonymity game, the verification oracle also prevents trivial distinguishing attacks against the challenge signature. If the game is already in attack mode, and the adversary asks to verify the challenge pseudonym (m^*, σ^*), then the oracle checks whether the trust graph contains exactly one of the two possible challenge trust relations. In that case, the adversary could distinguish the challenge signer only from the trust relation, rather than from the signature itself. Therefore, the oracle returns $\perp$ instead of answering the query. In all other cases, the oracle returns the real verification result b.

The validity checker ShouldVerify. The function $ShouldVerify$ defines the verification condition used by the security game. It does not run the cryptographic verification algorithm. Instead, it checks whether a message-signature pair should be accepted according to the history of the game.

The function first checks whether (m, σ) appears in the set S_{valid} of honestly generated signatures. If it does, then this step extracts the signer j by matching. Next the game locates the set T_j of users trusted by the signer j, i.e. the set of users who has user j's public identity key. This set is built by searching all the trust relations G_{trust}:

$$T_j = \{k : j \to k \in G_{trust}\}.$$

If a verifier ever trusts a corrupted user, then the function returns 1. This case captures the scenario that, once the identity keys of the signer are known, the adversary can verify the corresponding signatures.

If the signature was not honestly generated using the signing oracle and no trusted corrupted user explains its validity, then the function returns 0. Otherwise, if the signature was honestly generated by the signer j, the function checks whether the signer j is trusted

Algorithm 8: $\mathcal{O}_{verify}$

Oracle $\mathcal{O}_{verify}(m, \sigma, i, t)$: **begin**
  $step \leftarrow t - P.i.t$;
  $P.i.PK' \leftarrow \mathsf{FAMB.TRatchet}(P.i.PK, step)$;
  $b \leftarrow \mathsf{FAMB.Verify}(P.i.PK', m, \sigma)$;
  $b' \leftarrow \mathsf{ShouldVerify}(m, \sigma, i, t)$;
  if $b = 1$ ***and*** $b' = 0$ **then**
    $\mathsf{Abort}(\text{“Soundness Failure”})$;
  end
  if $b = 0$ ***and*** $b' = 1$ **then**
    $\mathsf{Abort}(\text{“Correctness Failure”})$;
  end
  if $mode = attack$ ***and*** $(m, \sigma) = (m^*, \sigma^*)$ ***and***
   $(chall_0 \rightarrow chall_1 \in G_{trust}$ ***xor*** $chall_1 \rightarrow chall_0 \in G_{trust})$ **then**
    return $\perp$;
  end
  return b;
end

by the verifier i. If the signer's public keys are known by the verifier, then the signature should verify, and the function returns 1. If i does not trust the signer j, then the signature should not verify, and the function returns 0.

The initialization oracle $\mathcal{O}_{init}$. The oracle $\mathcal{O}_{init}$ allows the adversary to create a new honest user. The oracle first assigns a fresh identity i to the new user. This identity is chosen as the next unused index, based on the current number of honest and corrupted users. The oracle then runs $\mathsf{FAMB.KeyGen}$ to generate the initial secret key and public identity key PK of the user. Note that, this so-called "public" identity key is actually hidden, and must be hidden in order to achieve the anonymity property. This is because in our context, the signature is used to identify who sends the messages instead of providing the message integrity. The resulting user state is stored together with the current time t, and the user is added to the set U_{honest}. Thus, after this oracle call, the adversary may interact with this user through the other oracles while not knowing the user's secret state.

Algorithm 9: ShouldVerify

Function $\mathsf{ShouldVerify}(m, \sigma, i, t)$: **begin**
  $b \leftarrow$ is $(m, \sigma, t, j) \in S_{valid}$ and $t \geq P.i.t$;
  $T_i \leftarrow \{k : k \rightarrow i \in G_{trust}\}$;
  if $T_i \cap U_{corrupt} \neq \varnothing$ **then**
    return 1;
  end
  if $b = 0$ **then**
    return 0;
  end
  if $j \in T_i$ **then**
    return 1;
  end
  return 0;
end

Algorithm 10: $\mathcal{O}_{init}$

Oracle $\mathcal{O}_{init}(t)$: **begin**
  $i \leftarrow |U_{honest}| + |U_{corrupt}| + 1$;
  $(SK_i, PK_i) \leftarrow \mathsf{FAMB.KeyGen}()$;
  $P \leftarrow (P.i.SK, P.i.PK, P.i.t) \leftarrow (SK_i, PK_i, t)$;
  $U_{honest}.\mathsf{add}(P.i)$;
  $P.i.L_{follow} \leftarrow \emptyset$;
  $P.i.L_{follower} \leftarrow \emptyset$;
end

The pseudonym generation oracle $\mathcal{O}_{pseugen}$. The oracle $\mathcal{O}_{pseugen}$ creates and publishes a pseudonym. The pseudonym itself is placed on a public append-only board. Its creator is recorded separately as private game metadata and is not revealed through the board.

Algorithm 11: $\mathcal{O}_{pseugen}$

Oracle $\mathcal{O}_{pseugen}(j, m, t)$: **begin**
  if $t < P.j.t$ **then**
    return $\perp$;
  end
  $(P', psu_j) \leftarrow \mathsf{FAMB.PseuGen}(P.j, t)$;
  $P.j \leftarrow P'$;
  $h \leftarrow \mathsf{FreshHandle}()$;
  $\mathcal{B}_{psu}[h] \leftarrow psu_j$;
  $\mathcal{M}_{psu}[h] \leftarrow (j, t)$;
  return (h, psu_j);
end

The signing oracle $\mathcal{O}_{sign}$. The oracle $\mathcal{O}_{sign}$ models honest signing queries. On input a message m, a user identity i, and a time t, the adversary can ask to obtain the corresponding signature. The oracle first checks whether the requested signing time is later than the current time stored in the user's state. If the requested time is not fresh, the oracle returns $\perp$. This prevents the adversary from forcing an honest user to sign at an old time.

If the time check succeeds, the oracle runs the signing algorithm FAMB.Sign using the secret key of user i. The resulting signature σ is added to S_{valid} together with the message, the time, and the identity of the signer. This record is later used by $ShouldVerify$ to decide whether the signature should be accepted. Finally, the oracle returns the signature σ to the adversary.

The time oracle $\mathcal{O}_{time}$. The oracle $\mathcal{O}_{time}$ models the passage of time for an honest user. On input a time t and a user identity i, the oracle first checks whether the user is corrupted. If the user is corrupted, the oracle does nothing, since corrupted users are controlled by the adversary and might not follow the standard time updates. The oracle also does nothing if the user's current time is already greater than or equal to t.

If t is a later time, the oracle advances the user's state to time t. It does this by running the signing-key ratchet FAMB.SRatchet on the user's current public identity key and secret

Algorithm 12: $\mathcal{O}_{sign}$

Oracle $\mathcal{O}_{sign}(m, i, t)$: **begin**
  if $t \leq P.i.t$ **then**
    return $\perp$;
  end
  $\sigma \leftarrow \mathsf{FAMB.Sign}(P.i.SK, m, t)$;
  $S_{valid}.\mathsf{add}((m, \sigma, t, i))$;
  return σ;
end

key, and by running the trust-state ratchet FAMB.TRatchet on the user's local state so that all the identity keys stored inside this user get updated as well. The user's secret key, local state, and time are then updated. This oracle captures the forward evolution of an honest user's state over time.

Algorithm 13: $\mathcal{O}_{time}$

Oracle $\mathcal{O}_{time}(t, i)$: **begin**
  if $i \in U_{corrupt}$ **then**
    return;
  end
  if $P.i.t \geq t$ **then**
    return;
  end
  $t' \leftarrow t - P.i.t$;
  $(PK', SK') \leftarrow \mathsf{FAMB.SRatchet}(P.i.PK, P.i.SK, t')$;
  $PK' \leftarrow \mathsf{FAMB.TRatchet}(P.i.P, t')$;
  $P.i.SK \leftarrow SK'$;
  $P.i.PK \leftarrow PK'$;
  $P.i.t' \leftarrow t'$;
  return;
end

The corruption oracle $\mathcal{O}_{corrupt}$. The oracle $\mathcal{O}_{corrupt}$ gives the adversary adaptive corruption capabilities. On input a user identity i, the oracle first checks whether i is currently

honest. If i is already corrupted, the oracle returns $\perp$.

In the anonymity game, additional restrictions are imposed after the challenge has been issued. If the game is in attack mode, the oracle rejects certain corruption queries that would allow the adversary to trivially distinguish the challenge signer. In particular, if user i is still at or before the challenge time and has a trust relation to one of the two challenge users, then the oracle returns $\perp$. This prevents the adversary from corrupting a user whose state would directly reveal information about the challenge relation.

If the corruption query is allowed, the oracle removes i from U_{honest} and adds it to $U_{corrupt}$. It then returns the full state of the user, including the public key, secret key, local state, and current time. From this point on, the user is considered corrupted.

Algorithm 14: $\mathcal{O}_{corrupt}$

Oracle $\mathcal{O}_{corrupt}(i)$: **begin**
  if $i \notin U_{honest}$ **then**
    return $\perp$;
  end
  if $mode == attack$ **then**
    if $P.i.t \leq chal.t$ ***and*** $(i \to chall.P_0 \in G_{trust}$ ***or*** $i \to chall.P_1 \in G_{trust})$ **then**
      return $\perp$;
    end
  end
  $U_{honest}.remove(i)$;
  $U_{corrupt}.add(i)$;
  return $P.i$;
end

The follow and allowfollow oracle $\mathcal{O}_{follow}$ and $\mathcal{O}_{allowfollow}$ Recall from Anix trust establishment that building one-way trust requires back-and-forth interactions. Suppose Alice follows Bob. First, Alice processes Bob's pseudonym and creates a follow message. Then Bob receives this message, decrypts it, and stores Alice's public identity key in his follower list. Therefore, we model this interaction using two oracles. The oracle $\mathcal{O}_{follow}$ models the sender side of the interaction for trust establishment. The function FindHandle traverses $\mathcal{M}_{psu}$ on (j, t) to find h. And then it lets user i follow user j by running the local follow algorithm of user i. The oracle returns the message that should be sent to j. The

oracle $\mathcal{O}_{allowfollow}$ models the receiver side. It lets user j process the follow message and update its follower list.

Algorithm 15: $\mathcal{O}_{follow}$

```
Oracle O_follow(i, j, t): begin
    h ← FindHandle(M_psu, j, t);
    if h = ⊥ then
        return ⊥;
    end
    psu ← B_psu[h];
    (P', m') ← FAMB.Follow(P.i, psu);
    P.i ← P';
    P.i.L_follow.add(j);
    return m';
end
```

The challenge oracle $\mathcal{O}_{challenge}$. The oracle $\mathcal{O}_{challenge}$ is used in the anonymity game. It takes two honest users i and j, a message m, and a challenge time t as input. The oracle first checks that both users are honest. If either user is not honest, the oracle returns $\perp$. It also checks that the challenge time is not earlier than the current time of either challenge user. This ensures that the challenge is generated at a valid future time.

The oracle can only be called once. If the game is already in attack mode, then a challenge has already been issued, and the oracle returns $\perp$. Otherwise, the oracle stores the two challenge users, the challenge message, and the challenge time. It then switches the game to attack mode and samples a hidden bit $chall.b \leftarrow \{0, 1\}$. This bit determines which of the two users is used to generate the challenge signature. The oracle then computes two signatures on the same message and at the same time: one using the secret key of user i, and one using the secret key of user j. Both signatures are added to S_{valid}, since both were honestly generated. The oracle returns only one of them, namely $\sigma_{chall.b}$. The adversary's goal in the anonymity game is to decide whether the returned signature came from user i or from user j.

Challenge state. The challenge state records the values associated with the anonymity challenge. The variables $chall.P_0$ and $chall.P_1$ store the two challenge users. The variable

Algorithm 16: $\mathcal{O}_{allowfollow}$

Oracle $\mathcal{O}_{allowfollow}(j, m, t)$: **begin**
  if $mode == attack$ **then**
    if $P.i.t \leq chal.t$ ***and*** $(i \rightarrow chall.P_0 \in G_{trust}$ ***or*** $i \rightarrow chall.P_1 \in G_{trust})$ **then**
      return $\perp$;
    end
  end
  if $j \notin P.i.L_{follow}$ **then**
    return $\perp$;
  end
  $(P', m') \leftarrow \mathsf{FAMB.AllowFollow}(P.j, m)$;
  $P.j \leftarrow P'$;
  $G_{trust}.\mathsf{add}(i \rightarrow j)$;
  $P.j.L_{follower}.\mathsf{add}(P.i)$;
end

Algorithm 17: $\mathcal{O}_{challenge}$

Oracle $\mathcal{O}_{challenge}(i, j, m, t)$: **begin**
  if $i \notin U_{honest}$ ***or*** $j \notin U_{honest}$ **then**
    return $\perp$;
  end
  if $t < P.i.t$ ***or*** $t < P.j.t$ **then**
    return $\perp$;
  end
  if $mode = attack$ **then**
    return $\perp$;
  end
  $chall.P_0, chall.P_1, chall.m, chall.t \leftarrow (i, j, m, t)$;
  $mode \leftarrow attack$;
  $chall.b \leftarrow \{0, 1\}$;
  $\sigma_0 \leftarrow \mathsf{FAMB.Sign}(P.i.SK, m, t)$;
  $\sigma_1 \leftarrow \mathsf{FAMB.Sign}(P.j.SK, m, t)$;
  $S_{valid}.\mathsf{add}((m, \sigma_0, t, i), (m, \sigma_1, t, j))$;
  return $\sigma_{chall.b}$;
end

$chall.m$ stores the challenge message, and $chall.t$ stores the challenge time. The hidden bit $chall.b$ determines which challenge user was used to produce the returned signature.

Algorithm 18: challenge state

$chall.P_0$;
$chall.P_1$;
$chall.P_\sigma$;
$chall.m$;
$chall.t$;
$chall.b$;

3.2 Instantiation

In this section, we instantiate algorithms in Definition 1 for FAMB. Note that we use Ed25519 in this instantiation because, as reviewed in Section 2.3, its key-blinded variant has been shown to satisfy unlinkability. However, the FAMB formalism is not tied to Ed25519 and may be instantiated with another standardized digital-signature scheme, provided that the scheme supports compatible key blinding and satisfies the required unlinkability and unforgeability properties.

FAMB.KeyGen. The algorithm FAMB.KeyGen instantiates the key generation algorithm 3 using the Ed25519-IETF key generation procedure described in Section 2.3. Here sk_{seed}

Algorithm 19: FAMB.KeyGen

$sk_{seed} \leftarrow \{0,1\}^{256}$;
$h \leftarrow \mathsf{SHA512}(sk_{seed})$;
$x \leftarrow \mathsf{ScalarClamp}(h[0:32])$;
$pre \leftarrow h[32:64]$;
$PK \leftarrow xG$;
$SK \leftarrow (sk_{seed}, x, pre)$;
return (SK, PK);

is the Ed25519 private seed, x is the secret scalar used for signing, and pre is the prefix

used in deterministic signing. In a concrete implementation, storing the seed is enough because x and pre can be recomputed from it. The public key PK is the Ed25519 public identity key. Recall that although it is called a public key, Anix treats it as a secret, as it identifies users.

FAMB.Sign. The algorithm FAMB.Sign instantiates signing using Ed25519-IETF with key blinding, as described in Section 3.2.2 Algorithm 6 and Algoirthm 3. On input a secret key SK, a message m, and a protocol time t, it first derives a blinding factor from the message and the corresponding public identity key, and uses the blinded signing key to generate the signature. The output is a blinded signature σ. In Anix, a pseudonym is composed of this signature concatenated with two one-time-use public keys. In this algorithm, we abstract away these two keys and focus on the signature component. The reason is that the one-time-use keys are sampled independently for each pseudonym and do not depend on the user's long-term identity key. Assuming they are not reused, they do not by themselves contribute to linkability or forgeability. Thus, it is sufficient to treat the blinded signature as the object for which we prove unlinkability and unforgeability. Note that we will still add two one-time-use public keys back when broadcasting as pseudonyms so that in the trust establishment phase, other user can use the one-time key for following.

Algorithm 20: FAMB.Sign

Data: secret key SK, message m, protocol time t
Result: blinded signature σ
$PK \leftarrow \mathsf{Pub}(SK)$;
$PK', SK' \leftarrow \mathsf{FAMB.SRatchet}(PK, SK, t)$;
$bf \leftarrow \mathsf{Hash}(m \parallel PK')$;
$\widetilde{SK'} \leftarrow \mathsf{Hash}(bf, SK') \cdot SK'$;
$\sigma \leftarrow \mathsf{Ed25519\text{-}IETF.Sign}(\widetilde{SK'}, m)$;
return σ;

FAMB.PseuGen. The algorithm FAMB.PseuGen constructs a complete pseudonym from the signature produced by FAMB.Sign. It first generates fresh one-time-use encryption and signing key pairs. It then signs the concatenation of the two one-time-use public keys using the user's current identity signing key.

Algorithm 21: FAMB.PseuGen

Data: User state P_i, protocol time t
Result: Updated user state P_i' and pseudonym psu
$(SkE_{OTU}, PkE_{OTU}) \leftarrow \mathsf{HPKE.KEM.KeyGen}()$;
$(SkS_{OTU}, PkS_{OTU}) \leftarrow \mathsf{Ed25519\text{-}IETF.KeyGen}()$;
$m_{psu} \leftarrow PkE_{OTU} \parallel PkS_{OTU}$;
$\sigma \leftarrow \mathsf{FAMB.Sign}(P_i.SK, m_{psu}, t)$;
$psu \leftarrow (PkE_{OTU}, PkS_{OTU}, \sigma)$;
$P_i' \leftarrow P_i$;
$P_i'.K_{OTU}[psu] \leftarrow (SkE_{OTU}, SkS_{OTU}, t)$;
return (P_i', psu);

FAMB.Verify. The algorithm FAMB.Verify is the verification counterpart of FAMB.Sign. On input a public identity key PK, a message m, and a signature σ, the verifier first recomputes the same blinding factor from the message and the public identity key, and then uses this blinding factor to derive the corresponding blinded public key bpk. Finally, it runs the Ed25519-IETF verification algorithm on bpk, m, and σ as .

Algorithm 22: FAMB.Verify

Data: public key PK, message m, signature σ,
$bf \leftarrow \mathsf{Hash}(m \parallel PK)$;
$bpk \leftarrow$ Ed25519-IETF.BlPubKey;
return $\mathsf{Ed25519\text{-}IETF.Verify}(PK, m, \sigma)$

FAMB.TRatchet. The algorithm FAMB.TRatchet instantiates the trust ratchet. This algorithm is used when a user stores another user's public identity key in its follower list and needs to update this stored key forward in time.

Here H is a hash function, and $r \cdot PK$ denotes scalar multiplication of the Ed25519 public key PK by the scalar r. The protocol time t records the time to which the stored key is updated. Since FAMB.TRatchet can be computed using only the public identity key, so followers can update their stored view of another user's key locally.

FAMB.SRatchet. The algorithm FAMB.SRatchet instantiates the signing-key ratchet. It updates both the secret identity key and the corresponding public identity key. In this

Algorithm 23: FAMB.TRatchet

Data: Public identity key PK, number of ratchet steps t
Result: Updated public identity key PK'
$PK^{(0)} \leftarrow PK$;
for $k \leftarrow 0$ **to** $t-1$ **do**
  $r_k \leftarrow H(PK^{(k)})$;
  $PK^{(k+1)} \leftarrow r_k \cdot PK^{(k)}$;
end
$PK' \leftarrow PK^{(t)}$;
return PK';

instantiation, we view the secret key SK as the expanded Ed25519 signing state containing the secret scalar x. The algorithm first derives a scalar ratchet factor from the current public key. It then multiplies the secret scalar by this factor modulo the Ed25519 group order L, and applies the same factor to the public key by scalar multiplication. This ensures that the updated secret key and public key still correspond to each other.

Here H is a hash function. The operation $r \cdot x \bmod L$ is multiplication in the scalar field, while $r \cdot PK$ is scalar multiplication of the Ed25519 public key. Thus, if the old public key satisfies $PK = xG$, then the updated public key satisfies

$$PK' = x'G.$$

This is the key consistency property needed for the ratcheting construction. After this step, users can use the ratcheted secret key to generate signatures.

FAMB.Follow. The algorithm FAMB.Follow instantiates the sender side of the one-way follow procedure. Suppose Alice wants to follow Bob after seeing Bob's pseudonym. The input to the algorithm is Alice's current local state P_A and Bob's pseudonym psu_B. The algorithm first parses Bob's pseudonym to obtain Bob's one-time-use public encryption key PkE_{OTU_B}. It then encrypts Alice's public identity key PubK_{ID_A} under this one-time-use public key. The output message OWTMessage is then sent through the mesh network to Bob. The algorithm also updates Alice's follow list, recording that Alice has chosen to follow Bob. We can instantiate the public-key encryption used in this algorithm with HPKE in base mode, using X25519 as the KEM [3].

Algorithm 24: FAMB.SRatchet

Data: Public identity key PK, secret identity key SK, number of ratchet steps t
Result: Updated key pair (PK', SK')
Parse SK as $(x^{(0)}, pre)$;
$PK^{(0)} \leftarrow PK$;
for $k \leftarrow 0$ **to** $t-1$ **do**
  $r_k \leftarrow H(PK^{(k)})$;
  $x^{(k+1)} \leftarrow r_k \cdot x^{(k)} \bmod L$;
  $PK^{(k+1)} \leftarrow r_k \cdot PK^{(k)}$;
end
$PK' \leftarrow PK^{(t)}$;
$SK' \leftarrow (x^{(t)}, pre)$;
return (PK', SK');

Algorithm 25: FAMB.Follow

Data: Alice's local state P_A, Bob's pseudonym psu_B
Result: Updated local state P'_A and follow message OWTMessage
Parse psu_B as $(\mathtt{PkE}_{\mathtt{OTU}B}, \sigma_B)$;
$\mathsf{PubK}_{ID_A} \leftarrow P_A.PK$;
$\mathsf{OWTMessage} \leftarrow \mathsf{Enc}_{\mathsf{PkE}_{OTU_B}}(\mathsf{PubK}_{ID_A})$;
$P'_A \leftarrow P_A$;
$P'_A.L_{follow} \leftarrow P_A.L_{follow} \cup \{psu_B\}$;
return $(P'_A, \mathsf{OWTMessage})$;

FAMB.AllowFollow. The algorithm FAMB.AllowFollow instantiates the receiver side of the one-way follow procedure. Suppose Bob receives a follow message from Alice. The input to the algorithm is Bob's current local state P_B and the received message OWTMessage. Bob uses the one-time-use secret encryption key corresponding to the pseudonym that Alice followed to decrypt the message. If decryption succeeds, Bob obtains Alice's public identity key PubK_{ID_A} and stores it in his follower list. From this point on, Bob can verify whether a future pseudonym was generated by Alice.

Algorithm 26: FAMB.AllowFollow

Data: Bob's local state P_B, follow message $\mathsf{OWTMessage}$
Result: Updated local state P'_B and optional response m'
$\mathsf{PubK}_{ID_A} \leftarrow \mathsf{Dec}_{\mathsf{SkE}_{OTU_B}}(\mathsf{OWTMessage})$;
if $\mathsf{PubK}_{ID_A} = \perp$ **then**
  return $(P_B, \perp)$;
end
$P'_B \leftarrow P_B$;
$P'_B.L_{follower} \leftarrow P_B.L_{follower} \cup \{\mathsf{PubK}_{ID_A}\}$;
$m' \leftarrow \epsilon$;
return (P'_B, m');

Here SkE_{OTU_B} is the one-time-use secret encryption key generated by Bob when he created the pseudonym that Alice used. In our instantiation, the decryption operation is implemented using the corresponding HPKE opening operation. The value $m' = \epsilon$ means that no further protocol message is needed in the basic one-way follow procedure.

Chapter 4

Key Ratcheting

4.1 Forward Anonymity

In Anix's design, anonymity is mainly achieved by generating a fresh pseudonym from the user's public key for each message. However, one issue is that, even though the pseudonym is fresh every time, if a user is coerced into revealing their public identity key, then all of their previous broadcasts can be linked to the user. To address this issue, we introduce a key ratcheting component to ensure forward anonymity.

Note that although this key ratcheting is related in spirit to the key ratcheting used in Signal, we cannot directly use Signal's ratchet in Anix. Signal's Double Ratchet is designed for a two-party encrypted conversation. After an initial shared secret is established, Signal continuously derives fresh message keys for sent and received messages by advancing a KDF chain for each message. When a party receives a new ratchet public key from the other party, it performs a new Diffie–Hellman computation and mixes the result into the root key to refresh the session state [14].

The ratchet needed for Anix lives in a different setup. Anix does not derive symmetric encryption keys for a pairwise session. And the value being updated is the identity key used to generate and recognize pseudonyms. Moreover, because of the way Anix designs its trust management, the ratchet must also update the public identity keys stored by followers, possibly while some users are out of sync. The goals are also different. Signal's ratchet protects the confidentiality of messages inside an ongoing conversation, while the Anix ratchet protects the unlinkability of pseudonyms over time.

Our design of key ratcheting can be used as follows. For each user, ratcheting is performed locally and does not require interaction with other users. Before generating a

signature, a user advances its identity key from its current ratchet time to a chosen signing time t using FAMB.SRatchet, and signs using the resulting identity key. The signing time t is attached to the pseudonym as a public time tag and is included in the signed data, so that it cannot be modified without invalidating the signature. After signing, the user may advance the ratchet by further steps and store only the latest key state. The intermediate and previous secret keys should be erased after use. Thus, the key used for a particular pseudonym need not be the same key that remains in the user's local state after the pseudonym is broadcast.

A follower stores the user's public identity key together with its corresponding ratchet time. To identify a received pseudonym tagged with time t, the follower first advances its stored public key to time t using FAMB.TRatchet, and then attempts verification. More precisely, if the stored key corresponds to time $t_0 \leq t$, the follower performs exactly $t - t_0$ ratchet steps before verifying the signature. This avoids repeatedly guessing the appropriate key and prevents an unbounded verification loop. If verification succeeds, the follower may update its stored public key and ratchet time. This also allows a user who was temporarily offline to catch up without contacting the signer. If a delayed message has a signing time earlier than the follower's current ratchet time, the follower cannot derive the earlier key by ratcheting backwards. Such messages may be supported by retaining a bounded cache of recent public keys; otherwise, they are rejected as stale. Each ratchet step requires only a hash-to-scalar computation and one elliptic-curve scalar multiplication. Nevertheless, the accepted difference between the stored time and the message time should be bounded, so that an adversary cannot attach an excessively large future time and force the verifier to perform an unreasonable number of ratchet steps.

Suppose an adversary learns a user's current public identity key. Because the ratchet is one-way, as we formally prove below, the adversary should not be able to recover the earlier public keys used for previous pseudonyms. Consequently, the current compromise does not reveal which earlier pseudonyms were generated by that user, providing forward anonymity.

4.1.1 Key-Ratcheting Scheme

A ratcheting scheme is an algorithm that takes a value and outputs a new value. In the context of Anix, the input value can be public key or private key.

4.1.2 Security Game

The security intuition is one-wayness. Note that we choose a one-wayness-based security definition to align with the assumption for the key ratcheting component in the Tamarin model as described in Chapter 5. In the Tamarin model, key ratcheting is described by rules that support forward key evolution but do not allow earlier key states to be recovered. We specify the security game of this ratcheting scheme in Algorithm 27.

Algorithm 27: One-Wayness Game

Pick s_0 uniformly at random;
The adversary chooses i;
The challenger picks a random oracle O;
for $k \leftarrow 1$ **to** i **do**
  $s_k \leftarrow s_{k-1} \cdot O(s_{k-1})$;
end
Send s_i to the adversary;
The adversary may query O adaptively;
The adversary outputs a guess s' for s_{i-1};
if $s_i = s' \cdot O(s')$ ***and*** s' *can be derived from* s_0 *by* $i-1$ *steps of ratcheting* **then**
  Adversary wins and the game aborts;
end

Each game starts with a randomly chosen seed s_0. First, the adversary picks i. Next, the challenger picks a random oracle O and performs the ratcheting i times using O until it gets s_i and returns s_i to the adversary. The winning condition for the adversary is to output a guess for s_{i-1}. During the game, the adversary can make query y to the random oracle, and the random oracle returns $O(y)$.

Note that the adversary wins the game if and only if s_{i-1} meets two requirements:

- $s_i = s_{i-1} \cdot O(s_{i-1})$
- more importantly, s_{i-1} can be derived from the seed s_0 with i steps of ratcheting

where the operation $\cdot$ denotes scalar multiplication over the corresponding group.

4.1.3 Construction

The construction of this key-ratcheting scheme is simple. We work in the Ed25519 group with generator P and group order L as Anix is instantiated based on this. Here $\times$ denotes scalar multiplication in the Ed25519 group. Secret keys are scalars in $\mathbb{Z}_L$, and public keys are group elements. As specified in Algoirthm 28, we make the updated key as a hash of the previous key multiplying by the previous key.

Algorithm 28: Ratcheting

Data: PK_i
Result: PK_{i+1}
$PK_{i+1} = PK_i \times H(PK_i)$;

4.1.4 Proof of Correctness

We now define how secret key is ratcheted and prove the key pairs still work together after ratcheting. Anix uses the Ed25519-IETF signing scheme. So let's define $sk = s_0 = \alpha$ and $pk = x_0 = \alpha P$. Then the ratcheted secret key is

$$s_0 \to s_1 = s_0 \cdot h(x_0) \to s_2 = s_1 \cdot h(x_1) \to ...s_k = s_{k-1} \cdot h(x_{k-1}) \tag{4.1}$$

We prove $s_k P = x_k$ by induction. The base case is simple. When $k = 0$, we have $s_0 P = \alpha P = x_0$. For the inductive step, we assume $s_k P = x_k$ and aim to prove $s_{k+1} P = x_{k+1}$. By definition, we have $s_{k+1} = s_k \cdot h(x_k)$ and $x_{k+1} = s_k \cdot h(x_k)$. So $lhs = s_k h(x_k) P = (s_k P) h(x_k) = x_k \cdot h(x_k) = x_k = rhs$.

4.1.5 Proof of Security

We now prove that the construction in Algorithm 28 satisfies the one-wayness definition in Algorithm 27. Let Real denote the experiment in Algorithm 27, and write $t = i$ for the number of ratchet steps chosen by the adversary. We use x_k to denote the successive ratchet states, so that $x_k = s_k$ in Algorithm 27.

Winning Real requires the adversary to output the particular predecessor x_{t-1} of the given state x_t. We count the adversary's final candidate s' as one additional random-oracle

query. This is without loss of generality as after producing s', the adversary may query the random oracle at s' before returning it. If the candidate is correct, this additional query is made at the hidden point x_{t-1}. In the following, N denotes the total number of adversarial random-oracle queries including this final query.

The game sequence. We define games $G_0, \ldots, G_N$. In every game, the challenger first samples an endpoint x_t uniformly from the oracle domain D, where $|D| = M$. The challenger then samples distinct points $x_0, \ldots, x_{t-1}$ and programs the random oracle so that

$$Q[x_k] = x_{k+1} \qquad \text{for every } 0 \leq k < t.$$

The difference between the games is the stage at which this hidden chain is sampled. In G_j, the chain is sampled after the first j adversarial query input-output pairs have been fixed. Thus, G_0 samples the chain before any adversarial query, whereas G_N samples it only after all N adversarial queries.

Once the chain has been sampled, the adversary wins if it queries one of the hidden points $x_0, \ldots, x_{t-1}$. The endpoint x_t is not included in this winning condition because it is already given to the adversary. The games are specified formally in Algorithm 29.

Algorithm 29: Game G_j

Pick x_t uniformly at random from D;
Initialize $Q \leftarrow \emptyset$;
$sampled \leftarrow$ false;
if $j = 0$ **then**
  Sample distinct points $x_0, x_1, \ldots, x_{t-1}$ uniformly at random without replacement from $D \setminus \{x_t\}$;
  for $k \leftarrow 0$ **to** $t - 1$ **do**
    $Q[x_k] \leftarrow x_{k+1}$;
  end
  $sampled \leftarrow$ true;
end
Oracle $RO_j(x, \ell)$: **begin**
  if $sampled =$ true ***and*** $x \in \{x_0, \ldots, x_{t-1}\}$ **then**
    Adversary wins and the game aborts;
  end
  if $x \in \mathrm{dom}(Q)$ **then**
    $y \leftarrow Q[x]$;
  end
  else
    Sample y uniformly at random from D;
    $Q[x] \leftarrow y$;
  end
  if $\ell = j$ ***and*** $sampled =$ false **then**
    Sample distinct points $x_0, x_1, \ldots, x_{t-1}$ uniformly at random without replacement from

$$D \setminus \big(\{x_t\} \cup \mathrm{dom}(Q) \cup \mathrm{range}(Q)\big);$$

    for $k \leftarrow 0$ **to** $t - 1$ **do**
      $Q[x_k] \leftarrow x_{k+1}$;
    end
    $sampled \leftarrow$ true;
  end
  return y;
end

Theorem 1 (One-wayness of the key-ratcheting scheme)**.** *Let $\mathcal{A}$ be an adversary in the one-wayness experiment of Algorithm 27. Suppose that $\mathcal{A}$ chooses $i = t \geq 1$ and makes at most N random-oracle queries, including the query on its final output. Let D be the oracle domain, with $|D| = M$, and assume that $M - 2N \gg t$.*

Conditioned on the ratchet chain containing no loop, Real *and G_0 are identically distributed. Moreover,*

$$\Delta(G_0, G_N) \leq \frac{2Nt}{M - 2N + 1},$$

and the adversary cannot win the ideal game G_N. Consequently,

$$\Pr[\mathcal{A} \textit{ wins Algorithm 27}] \leq \frac{t(t+1)}{2M} + \frac{2Nt}{M - 2N + 1}. \tag{4.2}$$

If N and t are polynomial in the security parameter and M is exponential, this probability is negligible. Therefore, the construction in Algorithm 28 is one-way according to the security definition in Algorithm 27.

Proof. We prove the theorem in four steps.

From Algorithm 27 to G_0. In Real, the states $x_0, x_1, \ldots, x_t$ are generated successively by evaluating the random oracle. In contrast, G_0 directly samples a sequence of distinct points and programs the oracle to form the corresponding chain. Conditioned on the states in Real being distinct, the two experiments have the same distribution.

The experiments can therefore differ only if the chain generated in Real contains a loop. When x_i is generated, there are at most i earlier chain points with which it may collide. Hence,

$$\Pr[\text{loop in the chain}] = 1 - \prod_{i=1}^{t} \frac{M - i}{M} \tag{4.3}$$

$$\leq \sum_{i=1}^{t} \frac{i}{M} \tag{4.4}$$

$$= \frac{t(t+1)}{2M}. \tag{4.5}$$

It follows that

$$\Delta(\mathsf{Real}, G_0) \leq \frac{t(t+1)}{2M}. \tag{4.6}$$

From G_0 to G_N. We next bound the statistical distance between consecutive games G_j and G_{j+1}. Fix $j \in \{0, \ldots, N-1\}$. Let A_j be the set of points available for sampling the hidden chain in G_j. The game G_{j+1} fixes one additional random-oracle query before sampling the chain. The input and output of this query each make at most one additional point unavailable. Let $B_j \subseteq A_j$ denote the set of these newly unavailable points. Therefore,

$$|B_j| \leq 2.$$

In G_j, the ordered tuple $(x_0, \ldots, x_{t-1})$ is sampled uniformly without replacement from A_j. In G_{j+1}, it is sampled uniformly without replacement from $A_j \setminus B_j$. We couple the two games by first sampling the tuple as in G_j. If the sampled tuple does not intersect B_j, the same tuple is also a valid sample in G_{j+1}, and the adversary receives the same oracle answers in both games. Thus,

$$\Delta(G_j, G_{j+1}) \leq \Pr\left[\{x_0, \ldots, x_{t-1}\} \cap B_j \neq \emptyset\right] \tag{4.7}$$

$$\leq 1 - \frac{(|A_j| - 2)_t}{(|A_j|)_t}, \tag{4.8}$$

where

$$(a)_t = a(a-1)\cdots(a-t+1)$$

denotes the falling factorial.

For every $a \geq t + 1$,

$$1 - \frac{(a-2)_t}{(a)_t} = 1 - \frac{(a-t)(a-t-1)}{a(a-1)} \tag{4.9}$$

$$= \frac{t(2a-t-1)}{a(a-1)} \tag{4.10}$$

$$\leq \frac{2t}{a}. \tag{4.11}$$

The endpoint x_t is reserved, and each of the first j queries makes at most two additional points unavailable. Therefore,

$$|A_j| \geq M - 2j - 1.$$

Substituting this bound into Equation 4.8 gives

$$\Delta(G_j, G_{j+1}) \leq \frac{2t}{M - 2j - 1}. \tag{4.12}$$

Applying the triangle inequality over all consecutive games, we obtain

$$\Delta(G_0, G_N) \leq \sum_{j=0}^{N-1} \Delta(G_j, G_{j+1}) \tag{4.13}$$

$$\leq \sum_{j=0}^{N-1} \frac{2t}{M - 2j - 1}. \tag{4.14}$$

Since

$$M - 2j - 1 \geq M - 2N + 1 \qquad \text{for every } j \in \{0, \ldots, N-1\},$$

it follows that

$$\Delta(G_0, G_N) \leq \frac{2Nt}{M - 2N + 1}. \tag{4.15}$$

The ideal game G_N. In G_N, the hidden chain is sampled only after all N random-oracle queries have been fixed. Moreover, its points are selected from outside both the domain and range of the existing oracle table. Therefore, none of the adversary's queries can be one of the hidden points $x_0, \ldots, x_{t-1}$.

The adversary's final candidate is included as its last query. Consequently, there is no separate unqueried guess to consider, and

$$\Pr[G_N \text{ wins}] = 0. \tag{4.16}$$

Combining the games. Combining Equations 4.6, 4.15, and 4.16, we obtain

$$\Pr[\mathcal{A} \text{ wins Algorithm 27}] \leq \Delta(\mathsf{Real}, G_0) + \Delta(G_0, G_N) + \Pr[G_N \text{ wins}] \tag{4.17}$$

$$\leq \frac{t(t+1)}{2M} + \frac{2Nt}{M - 2N + 1}. \tag{4.18}$$

This bound is negligible when N and t are polynomial in the security parameter and M is exponential.

It remains to relate the point-valued random-oracle model used in the games to the concrete construction in Algorithm 28. The concrete ratchet computes

$$PK' = H(PK) \cdot PK$$

.

For any fixed non-identity point PK in a prime-order group, the map

$$r \longmapsto r \cdot PK$$

is a bijection from the nonzero scalars to the non-identity group elements. Therefore, if $H(PK)$ is uniformly distributed over the nonzero scalars, then $H(PK) \cdot PK$ is uniformly distributed over the non-identity group elements. Random-oracle outputs on distinct inputs are independent, and applying a bijection separately to each output preserves both independence and uniformity. Hence, the induced function

$$PK \longmapsto H(PK) \cdot PK$$

has the same distribution as the point-valued random oracle used in the game sequence. The bound above therefore applies to the concrete ratchet in Algorithm 28, completing the proof. □

Chapter 5

Proving FAMB in Tamarin

A pen-and-paper proof would normally proceed by a reduction argument using the FAMB formalism and one-wayness of key ratcheting defined above. In particular, an adversary that breaks unlinkability or soundness would be transformed into an adversary that breaks the one-wayness of the key-ratcheting scheme. However, FAMB is a multi-party, stateful protocol involving adaptive corruption, trust establishment, ratcheting, and potentially out-of-sync key states. A complete reduction would therefore require considering a large number of interactions and cases. Thus, we explore whether Tamarin can automate the reasoning about these protocol-level properties.

In this chapter, we give a detailed explanation of the soundness proof and unlinkability proof attempts for FAMB in Tamarin.

5.1 Modeling Abstraction

The model does not encode every detail of FAMB precisely. Instead, it is an abstraction designed to reason about the signature-based soundness and unlinkability of the protocol. The following simplifications are used.

No modeling of mesh topology. We abstract away the topology of message broadcasting. In the real mesh network, the way a message propagates may reveal side-channel information, for example which users were close enough to receive or forward a message. This is not modeled here. The security properties considered here are only reasoned at the cryptographic level.

Trust establishment is collapsed into one rule. In practice, the trust establishment contains two sides. First, the follower creates a follow message containing its public identity key encrypted under the one-time-use encryption key from the followed user's pseudonym. Then the receiver decrypts the message and stores the follower's public identity key. In the Tamarin model, we collapse this back-and-forth interaction into a single `Trust` rule.

Endorsements are omitted. Finally, we do not model message endorsement separately. In Anix, an endorsement is generated using the same blinded signing mechanism as pseudonym generation with the same pair of keys. That is to say, endorsement does not introduce a different cryptographic primitive. If the adversary could forge endorsements, then the same issue would already appear as a forgery of the blinded signature scheme. Conversely, if the signature component is unforgeable, then adding endorsements does not create a new independent source of signature forgery.

5.2 soundness_std.spthy

This section explains how FAMB is modeled in Tamarin and how Tamarin proves the unforgeability lemma for this model.

5.2.1 Cryptographic primitives (L11–L13)

```
builtins: multiset
functions: sign/2, pk/1, verify/3, true/0, false/0
equations: verify(sign(m, k), m, pk(k)) = true()
```

The model first declares the symbolic primitives used for signatures and for time expressions. The `multiset` built-in is used because protocol time is represented by symbolic terms that can be combined with the operator `+`. For example, a later time may be written as `t + stept`. Note that it allows a different system to express the concept of time from Tamarin's trace-time indices such as `#i`, which are used only to order rule applications in a trace. We refer the time manipulated using the multiset builtin as protocol time.

The functions `sign/2`, `pk/1`, and `verify/3` model digital signatures. A term `sign(m,k)` represents a signature on message m under key k, and `pk(k)` represents the corresponding public verification key. The verification equation

$$verify(sign(m, k), m, pk(k)) = true()$$

states that a correctly generated signature verifies under the corresponding public key and message. Note that this implicitly says that for other cases, the result of verification is underspecified. Thus, the adversary can only produce a valid signature for a key if it knows the corresponding signing key.

5.2.2 Restrictions (L17–L24)

```
1  restriction ratchet_once:
2  "All id t lsk1 lpk1 lsk2 lpk2 #i #j.
3  Ratcheted(id, t, lsk1, lpk1) @ #i
4  & Ratcheted(id, t, lsk2, lpk2) @ #j
5  ==> lsk1 = lsk2 & lpk1 = lpk2"
6
7  restriction OnlyOnce_Corrupt_TrustedKey:
8  "All id1 id2 lpk1 trust_time #i #j.
9  OnlyOnce_Corrupt_TrustedKey(id1, id2, lpk1, trust_time) @ #i &
      OnlyOnce_Corrupt_TrustedKey(id1, id2, lpk1, trust_time) @ #j
      ==> #i = #j"
10
11 restriction OnlyOnce_Trust:
12 "All id1_lpk #i #j.
13 OnlyOnce_Trust(id1_lpk) @ #i
14 & OnlyOnce_Trust(id1_lpk) @ #j
15 ==> #i = #j"
```

Three global restrictions limit the searchable trace space:

`ratchet_once` makes sure that for any user id and time t, the pair (lsk, lpk) produced for the ratchet slot is unique. It guarantees that two $Ratcheted(id, t, \dots)$ events at the same (id, t) must agree on the keys. This makes the ratchet a *function* of (id, t) rather than a relation, so `!Ratchet` facts can be matched unambiguously across rules.

`OnlyOnce_Corrupt_TrustedKey` simply says that any user's key stored in trusted users' list can be corrupted at most once. It prevents redundant repeated corruption events that would otherwise blow up the search.

`OnlyOnce_Trust` means that the same pair of users can perform `Trust` in the same direction at most once.

Under the hood, Tamarin treats restrictions as trace constraints. Thus, during proof search, any branch that would lead to a trace violating one of these restrictions is ruled out, reducing the search space.

5.2.3 Key generation and user creation (L26–L37)

```
1  rule KeyGen: [Fr(~lsk)] --> [InitKeys(~lsk, pk(~lsk))]
2
3  rule CreateUser:
4      [Fr(~id), InitKeys(id_lsk, pk(id_lsk)), In(t)]
5      --[Ratcheted(~id, t, id_lsk, pk(id_lsk)),
6         HonestKey(pk(id_lsk))]->
7      [!User(~id, id_lsk, pk(id_lsk), t),
8       !Ratchet(~id, t, id_lsk, pk(id_lsk)),
9       Out(<~id, t>)]
```

The first two rules model the creation of honest users. The rule `KeyGen` generates a fresh secret key `~lsk` and outputs the corresponding public key `pk(~lsk)`. The result is stored in the linear fact `InitKeys`.

The rule `CreateUser` consumes an `InitKeys` fact and assigns the key pair to a fresh user identity `~id`. The time t is supplied by the channel through the premise `In(t)`. This means that the model allows the adversary to choose at which protocol time the user is created.

This rule has two action facts. The action fact

$$Ratcheted(\widetilde{id}, t, id_lsk, pk(id_lsk))$$

records that the key pair $(id_lsk, pk(id_lsk))$ belongs to the ratchet slot of user $\widetilde{id}$ at time t. Together with the restriction `ratchet_once`, this ensures that the key assigned to a fixed user-time pair is unique.

The action fact

$$HonestKey(pk(id_lsk))$$

marks the public key as an honestly generated key. This label is later used in the soundness lemma, where we only make unforgeability claims about keys that were generated honestly.

The conclusions of `CreateUser` generate two persistent facts. The fact `!User` records the user with the key pair at time t, while `!Ratchet` records the same key pair in the ratchet store, allowing the adversary to perform ratcheting later. The output `Out(<~id, t>)` publishes the user identity together with its time, so that later rules can refer to this user.

5.2.4 Ratchet (L39–L44)

```
rule Ratchet:
    [In(id), In(t), Fr(~lsk)]
    --[Ratcheted(id, t, ~lsk, pk(~lsk)),
       HonestKey(pk(~lsk))]->
    [!Ratchet(id, t, ~lsk, pk(~lsk))]
```

Ratchet rule is used to generate ratchet facts at arbitrary time points. On input an identity id and a time t, it generates a fresh secret key `~lsk` and stores the corresponding key pair in the persistent fact `!Ratchet`. As in user creation, the action fact `Ratcheted` records that this key pair belongs to the slot (id, t), and `HonestKey` records that the public key was honestly generated.

5.2.5 Adversarial Key Ratcheting (L46–L66)

Two symmetric rules allow the adversary to derive updated keys at desired timepoint from current secret or public key:

```
rule AdvRatchet:
    let wut = t + stept in
    [In(lsk), In(stept),
     !Ratchet(id, t, lsk, pk(lsk)),
     !Ratchet(id, wut, lsk_new, pk(lsk_new))]
    --[GetRatchet1(id, lsk_new), Reveal(pk(lsk_new))]->
    [Out(lsk_new)]

rule GetRatchetKey:
    let wut = t + stept in
    [In(pk(lsk)), In(stept),
     !Ratchet(id, t, lsk, pk(lsk)),
     !Ratchet(id, wut, lsk_new, pk(lsk_new))]
```

```
14      --[GetRatchet(id, pk(lsk_new))]->
15      [Out(pk(lsk_new))]
```

In both rules, the term

$$wut = t + stept$$

represents a later time and the value `stept` is supplied by the adversary. The rule `AdvRatchet` models ratcheting for secret keys. If the adversary already knows the secret key lsk at time t, and wants the key at time wut to be known by the adversary. It requires a ratchet fact at wut as an input, which enforces this fact to form in the previous rule Ratchet. The rule therefore outputs `lsk_new` at the end. It also emits the action fact

$$Reveal(pk(lsk_new))$$

which records that the secret key corresponding to the public key $pk(lsk_new)$ has been revealed to the adversary. Note that `Reveal` is indexed by the public key rather than the secret key due to a technical trick required for unforgeability lemma.

The rule `GetRatchetKey` models the public part of the same process. Note that this rule does not emit a `Reveal` action, because learning a public key is irrelevant in the soundness context.

5.2.6 Trust establishment (L68–L75)

```
1   rule Trust:
2       [In(<id1, uct1>), In(<id2, uct2>),
3        !User(id1, id1_lsk, pk(id1_lsk), uct1),
4        !User(id2, id2_lsk, pk(id2_lsk), uct2)]
5       --[TrustOK(id1, id1_lsk, pk(id1_lsk),
6                  id2, id2_lsk, pk(id2_lsk)),
7          OnlyOnce_Trust(pk(id1_lsk))]->
8       [Trust(id1, id2, pk(id1_lsk), uct1)]
```

The rule `Trust` models the establishment of a trust relation between two users. The first user, $id1$, is the user whose public key is stored, and the second user, $id2$, is the user who stores the key of $id1$. The two `!User` fact extract the user ids and key at specific time. And the conclusion

$$Trust(id1, id2, pk(id1_lsk), uct1)$$

represents that $id2$ has stored the public key of $id1$ at time $uct1$.

The action fact `TrustOK` records that the trust rule has been executed with these concrete users and keys, which is used for executability lemma. The action fact `OnlyOnce_Trust(pk(id1_lsk))` is used by the restriction `OnlyOnce_Trust`, which prevents the same trusted public key from being inserted repeatedly.

The resulting `Trust` fact is linear. This is intentional, as a trust fact is treated as a state cell that can later be advanced by `Tick_trust`. When the trusted user's key is ratcheted forward, the old trust fact is consumed and replaced by a new one containing the updated public key.

5.2.7 Ticking (advancing time) (L77–L98)

```
rule Tick:
    let wut = uct + stept in
    [In(stept), In(<id, uct>),
     !User(id, lsk, pk(lsk), uct),
     !Ratchet(id, wut, new_lsk, pk(new_lsk))]
    --[TickTime(uct, wut, stept)]->
    [!User(id, new_lsk, pk(new_lsk), wut),
     Out(<id, wut>)]

rule Tick_trust:
    let wut1 = uct1 + stept in
    [In(id2), In(id1), In(stept),
     Trust(id1, id2, id1_lpk, uct1),
     !Ratchet(id1, wut1, new_id1_lsk, pk(new_id1_lsk))]
    --[TickTrustOK(id1, id2, wut1)]->
    [Trust(id1, id2, pk(new_id1_lsk), wut1)]
```

The rule `Tick` advances a user's own key state. If user id has key pair $(lsk, pk(lsk))$ at time uct, and the ratchet fact contains a key pair $(new_lsk, pk(new_lsk))$ at the later time $wut = uct + stept$, then the rule creates a new `!User` fact for user id at time wut. The rule also outputs `<id, wut>` so that later rules can refer to the user at this updated time. Note that we choose to make `!User` a persistent fact. So each tick generates a new user with new key at a new time. This design decision is to make each fact of the form

`!User(id, lsk, pk(lsk), t)`

acts as a handle to the state of user id at protocol time t. This means that later rules are not forced to follow a single linear current state of the user. The adversary can choose a

user together with a protocol time, and then apply a rule to the corresponding historical state if that state exists in the model. For example, even after a user has been advanced to a later key, a corruption rule may still match an older `!User` fact and reveal the key at that earlier time. And we use restrictions on the adversary to ensure that a forgery is not valid if it matches a key that comes after such a retroactive corruption.

The rule `Tick_trust` advances the key stored in a truster's list. Suppose user $id2$ stores a trusted public key for user $id1$ at time $uct1$, using the ratchet fact to force a later key for $id1$ at time $wut1 = uct1 + stept$, then the trust entry is updated to contain the later public key $pk(new_id1_lsk)$. Unlike `!User`, the `Trust` fact is linear. Therefore, the old trusted key is consumed and replaced by the updated one. This separation between `Tick` and `Tick_trust` reflects the protocol structure. A user can advance its own key state, and another user's stored view of that key is advanced separately.

5.2.8 Corruption (L100–L112)

```
rule Corrupt_ownkey:
    [In(<id, uct>), !User(id, lsk, pk(lsk), uct)]
    --[CorruptPK1(id, uct), Reveal(pk(lsk))]->
    [Out(<pk(lsk), lsk, id, uct>)]

rule Corrupt_trustedkey:
    [In(id1), In(id2), Trust(id1, id2, id1_lpk, trust_time)]
    --[Corrupt_trustedkeyOK(id1, id1_lpk),
       OnlyOnce_Corrupt_TrustedKey(id1, id2, id1_lpk, trust_time
          )]->
    [Out(<id1, id1_lpk, trust_time>),
     Trust(id1, id2, id1_lpk, trust_time)]
```

The rule `Corrupt_ownkey` models a full corruption of a user's own key state. If the adversary selects user id at time uct, then the rule outputs the user's public key, secret key, identity, and time to the open channel. It also emits the action fact $Reveal(pk(lsk))$. This action fact records that the secret key corresponding to $pk(lsk)$ has been compromised. The soundness lemma later uses this fact as one of the two allowed explanations for a successful verification under an honest public key.

The rule `Corrupt_trustedkey` models a weaker form of corruption. It allows the adversary to inspect a public key stored in a user whom the key's owner trusts. The rule outputs the identity $id1$, the stored public key $id1_lpk$, and the time at which this trusted

key is valid to the channel. Since there is no secrete key stores in the trusters' key list, it does not output the corresponding secret key and does not emit a `Reveal` action.

The action fact `OnlyOnce_Corrupt_TrustedKey` is used by the corresponding restriction to prevent the same trusted-key entry from being corrupted repeatedly.

5.2.9 Signing and verification (L119–L135)

```
1  rule Sign:
2      let sig = sign(m, lsk_signed) in
3      [In(id), In(m), In(t),
4       !Ratchet(id, t, lsk_signed, pk(lsk_signed))]
5      --[Signed(sig)]->
6      [Out(sig)]
7
8  rule Verify:
9      [In(<m, sig, lpk>)]
10     --[_restrict(verify(sig, m, lpk) = true()),
11        VerifyOK(m, sig, lpk)]->
12     []
```

The rule `Sign` models honest signing. The adversary chooses a user identity id, a message m, and a symbolic time t. Similar as above, it uses Ratchet fact to force a generation of key at a desired time, and produces the symbolic signature $sig = sign(m, lsk_signed)$. In the end, the signature is output to the open channel.

The rule `Verify` models successful verification. It takes as input a message, a signature, and a public key. The embedded restriction $_restrict(verify(sig, m, lpk) = true())$ requires that the verification term reduces to `true()` in order for the rule to fire. By the signature equation declared earlier, this can only happen when the signature has the form `sign(m,k)` and the public key is `pk(k)` for the same key k. Thus, `Verify` fires exactly for signatures that are valid in the symbolic signature theory.

When verification succeeds, the rule emits the action fact $VerifyOK(m, sig, lpk)$. This action fact is the event used in the soundness lemma and executability lemma. It marks the point in the trace where the adversary has successfully verified a signature under a public key.

5.2.10 The unforgeability lemma (L145–L151)

```
lemma unforgeability:
"All m sig lpk #i.
    VerifyOK(m, sig, lpk) @ #i &
    (Ex #j. HonestKey(lpk) @ #j)
    ==>
    (Ex #k. Signed(sig) @ #k & #k < #i)
  | (Ex #k. Reveal(lpk) @ #k & #k < #i)"
```

The signing and verification rules should be understood together with the `unforgeability` lemma. `Sign` and `Verify` plays the role of the signing oracle and verification oracle in the classical unforgeability game.

The central soundness claim is stated in the lemma `unforgeability`. In logical form, it says that for every message m, signature sig, public key lpk, and trace position $\#i$, if verification succeeds at $\#i$ and lpk was ever generated as an honest key, then one of two things must have happened earlier in the trace:

$$\exists \#k.\ Signed(sig)@\#k \wedge \#k < \#i$$

or

$$\exists \#k.\ Reveal(lpk)@\#k \wedge \#k < \#i$$

In words, whenever the adversary successfully verifies a signature under an honest public key, then either the signature was honestly produced before the verification event, or the corresponding secret key was revealed before the verification event. This captures the intended soundness property of the signature part of the model. A valid signature under an honest key cannot appear from an unknown source. It must either come from an honest signing event or from a previous compromise of the signing key.

The `Reveal` events used by this lemma are produced by `Corrupt_ownkey` and `AdvRatchet`. The first rule models direct secret-key compromise, while the second models the forward derivation of later secrets after a previous secret has been learned.

5.2.11 Executability lemmas (L155–L196)

In the end, there are 9 `exists-trace` lemmas. These lemmas are not security properties in the same sense as `unforgeability`. Instead, they are sanity checks for the model. An `exists-trace` lemma asks Tamarin to find at least one trace in which a certain action fact can be found. These executability lemmas check that the main rules of the model are

reachable. For example, they show that the model admits traces in which a user can be created, a ratchet key can be formed, a signature can be generated, a signature can be verified, a trust relation can be established, time can be advanced, and corruption rules can be triggered. This is important because a security lemma can become trivially true if the protocol rules are unreachable.

5.2.12 Proofs Interpretation

Using Tamarin's builtin heuristic S (ordering constraints in the smart mode with no loop breaker delay) and auto-sourcing, Tamarin is able to autoprove the unforgeability lemma. The proof is shown below. Tamarin proofs are hard to parse, so we will work through it step-by-step in the rest of this section.

```
lemma unforgeability:
  all-traces
  "All m sig lpk #i.
    ((VerifyOK(m, sig, lpk) @ #i)
     & (Ex #j. HonestKey(lpk) @ #j))
    ==>
    ((Ex #k. Signed(sig) @ #k & #k < #i)
     | (Ex #k. Reveal(lpk) @ #k & #k < #i))"
/*
guarded formula characterizing all counter-examples:
"Ex m sig lpk #i.
    VerifyOK(m, sig, lpk) @ #i
  & (Ex #j. HonestKey(lpk) @ #j)
  & (All #k. Signed(sig) @ #k ==> not(#k < #i))
  & (All #k. Reveal(lpk) @ #k ==> not(#k < #i))"
*/
simplify
solve( VerifyOK( m, sig, lpk ) @ #i )
  case Verify___VARIANT_2
  solve( HonestKey( pk(x) ) @ #j )
    case CreateUser
    solve( InitKeys( x, pk(x) ) >>1 #j )
      case KeyGen
      solve( !KU( sign(m, ~lsk) ) @ #vk.3 )
        case Sign
        by contradiction
```

```
      next
        case c_sign
        solve( !KU( ~lsk ) @ #vk.6 )
          case AdvRatchet
          by contradiction
        next
          case Corrupt_ownkey_case_1
          by contradiction
        next
          case Corrupt_ownkey_case_2
          by contradiction
        qed
      qed
    qed
  next
    case Ratchet
    solve( !KU( sign(m, ~lsk) ) @ #vk.3 )
      case Sign
      by contradiction
    next
      case c_sign
      solve( !KU( ~lsk ) @ #vk.7 )
        case AdvRatchet_case_1
        by contradiction
      next
        case AdvRatchet_case_2
        by contradiction
      next
        case Corrupt_ownkey
        by contradiction
      qed
    qed
  qed
qed
```

The lemma is an `all-traces` lemma, so it must hold for every possible execution of the model. Tamarin proves this lemma by contradiction. It negates the lemma and searches for a trace that violates it. The negation of the lemma means that there exists a trace containing a successful verification event

$$VerifyOK(m, sig, lpk)@\#i$$

under a public key that was generated honestly, but where there is no earlier `Signed(sig)` event and no earlier `Reveal(lpk)` event.

Since Tamarin uses backward search, it first finds where the successful verification event came from. The only rule that can produce this action fact is `Verify`. Therefore, Tamarin reduces the constraint to the verification rule.

Next, as the verification rule contains the embedded restriction

$$verify(sig, m, lpk) = true(),$$

and the symbolic signature equation is

$$verify(sign(m, k), m, pk(k)) = true(),$$

Tamarin learns that the verified signature must have the form $sig = sign(m, x)$ and that the public key must have the form $lpk = pk(x)$ for some signing key x.

The next selected constraint is

```
solve( HonestKey( pk(x) ) @ #j )
```

This asks where the honest public key came from. In this model, there are only two sources of an `HonestKey` action. One is from `CreateUser`, where the initial key of a newly created user is marked honest, another comes from `Ratchet`, where a ratcheted key is generated and marked honest. Therefore, the proof splits into two main cases:

```
case CreateUser
...
next
  case Ratchet
...
```

In the `CreateUser` branch, Tamarin additionally resolves the source of the linear fact `InitKeys`. This fact can only be produced by `KeyGen`. Hence, this branch corresponds to the case where the verified public key is the initial key of an honest user. After the source of the honest key is fixed, Tamarin asks how the adversary could have obtained the signature term

$$sign(m, \texttt{~lsk}).$$

This appears in the proof as

```
solve( !KU( sign(m, ~lsk) ) @ #vk.3 )
```

The fact `!KU` denotes adversary knowledge. Thus, this step asks where the adversary's knowledge of the signature came from. There are two possible answers. First, the signature may have come from the honest signing rule

```
case Sign
by contradiction
```

This case contradicts the counterexample formula. The counterexample assumes that there is no earlier `Signed(sig)` event. However, if the signature came from the `Sign` rule, then such a `Signed(sig)` event does exist. Therefore, this branch cannot describe an attack.

Second, the adversary may have constructed the signature term by itself:

```
case c_sign
solve( !KU( ~lsk ) @ #vk.6 )
```

The case `c_sign` corresponds to the adversary applying the public constructor `sign`. In the symbolic model, the adversary can construct `sign(m,k)` only if it knows both m and k. Since the message can be known to the adversary, the remaining question is how the adversary learned the signing key `~lsk`. Tamarin therefore opens another source goal for

$$!KU(\texttt{\~{}lsk}).$$

The remaining proof shows that the signing key can only be obtained through ratchet compromise or direct corruption:

```
case AdvRatchet
by contradiction
next
  case Corrupt_ownkey_case_1
  by contradiction
next
  case Corrupt_ownkey_case_2
  by contradiction
```

Each of these cases emits a `Reveal` action for the corresponding public key. But the counterexample formula assumes that there is no earlier `Reveal(lpk)` event. Hence, all of these branches also close by contradiction.

The `Ratchet` branch follows the same logic. In that branch, the honest key comes from the ratchet population rule rather than from user creation. Tamarin again asks how the adversary obtained the valid signature term. Either the signature came from the honest `Sign` rule, which contradicts the absence of a previous `Signed(sig)` event, or the adversary constructed it using the signing key. In the latter case, the adversary must know the secret key, and the only possible sources are `AdvRatchet` or `Corrupt_ownkey`. Both imply an earlier `Reveal(lpk)` event, again contradicting the counterexample.

Since every possible source of the successful verification leads to a contradiction, Tamarin closes all branches of the proof tree. This proves that there is no trace in the model where a signature verifies under an honest public key unless the signature was previously signed honestly or the corresponding secret key was previously revealed. Therefore, this proof establishes the unforgeability property.

5.3 soundness.spthy

In the previous section, we used the standard symbolic model of digital signatures. In that model, signature verification is specified by the equation

$$verify(sign(m, k), m, pk(k)) = true().$$

Note that there are no other equations involving `verify`. This means that this equation specifies the only way in which signature verification can succeed. In other words, a signature verifies only against the exact message and the exact public key corresponding to the signing key used to produce it.

This model captures the intuition that a signature generated with key k should verify under $pk(k)$, and should not verify under any other public keys. It is the standard model of digital signatures used in many symbolic protocol analyses. However, this model is actually stronger than what many real signature schemes guarantee. Standard cryptographic notions such as EUF-CMA and SUF-CMA only rule out the ability of an adversary to forge a new valid signature under an honestly generated public key. But this property do not necessarily imply that verification can succeed only in the exact syntactic way expressed by the symbolic equation above.

This distinction matters for real-world signature schemes. For some schemes, there may exist unexpected verification behavior. For example, it may be possible that a signature verifies under a public key different from the one originally intended, or that an adversary can construct unusual public keys for which existing signatures verify. Some schemes also admit weak keys, message-key substitution behavior, or limited forms of signature malleability. All these behaviors are not captured by the standard modeling of digital signature.

Therefore, in this section we use an alternative model of digital signatures, called SVS, following the approach of Jackson, Cremers, Cohn-Gordon, and Sasse in *Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures* [10]. The goal of SVS is to model only the guarantees of the standard EUF assumptions, while leaving other verification behavior underspecified. In this way, the Tamarin model does not accidentally assume stronger properties than the signature scheme actually provides.

Concretely, the main implementation difference is that we no longer model verification as a function constrained by one equation. Instead, we model verification by using action facts and restrictions. The action fact records that a verification attempt occurred, including the signature, the message, the public key, and the result of verification. Restrictions then specify only the security properties that should always hold. In particular, when verification uses an honestly generated public key, the restrictions enforce correctness and unforgeability. For other cases, the model does not fully specify whether verification returns `true()` or `false()`, but leaves this decision to the modeler. Under the hood, the verification result is not computed by a `verify` equation. Rather, it is represented as input data in a `Verified` action.

Concretely, the SVS model uses an action of the following form:

$$Verified(sig, sigm, sigpk, verm, verpk, result).$$

The first three arguments describe the signature itself. The value *sig* is the signature term, *sigm* is the message for which the signature was originally produced, and *sigpk* is the public key corresponding to the signing key used to produce it. The next two arguments, *verm* and *verpk*, describe the message and public key used in the verification attempt. Finally, *result* records whether this verification should return `true()` or `false()` within the context of the the specific rule. This makes it possible to model attacks where a signature verifies against an unexpected key or message, while still enforcing the usual unforgeability guarantee for honestly generated public keys.

To formulate these restrictions, the model must be able to extract the original message and signing key from a signature term. This is done using two private helper functions:

```
functions: e1/1 [private], e3/1 [private]
equations: e1(sign(x,y,z)) = x
equations: e3(sign(x,y,z)) = z
```

The function `e1` extracts the message from a signature, while `e3` extracts the signing key. These functions are marked `private`, so the adversary cannot use them.

There are three restrictions to guarantee the basic properties of signature. The first one is correctness. It says that if a signature is honestly generated and verified against the same message and the same honest public key, verification must not return `false()`:

```
restriction Correctness:
"All sig m pk #i #j.
    Honest(pk) @ #i
  & Verified(sig, m, pk, m, pk, false) @ #j
  ==> F"
```

The second restriction is no-forgery. If verification succeeds under an honestly generated public key, then the verification message and verification key must match the message and public key for which the signature was originally produced:

```
restriction NoForgery:
"All sig sigm sigpk verm verpk #i #j.
    Honest(verpk) @ #i
  & Verified(sig, sigm, sigpk, verm, verpk, true) @ #j
  ==> sigm = verm & sigpk = verpk"
```

The third restriction is consistency. Verification is a deterministic algorithm. Therefore, if the same signature is verified against the same message and the same public key twice, the two verification results must be equal:

```
restriction Consistency:
"All sig sigm sigpk verm verpk result1 result2 #i #j.
    Verified(sig, sigm, sigpk, verm, verpk, result1) @ #i
  & Verified(sig, sigm, sigpk, verm, verpk, result2) @ #j
  ==> result1 = result2"
```

Despite this weaker modeling choice, the soundness lemma is still automatically proved by Tamarin using Smart heuristic. We will not walk through the proof here, as it is mostly similar to the previous proof.

5.4 Towards Proving Unlinkability

In this section, we describe our attempts towards proving the unlinkability property of the protocol in Tamarin. Compared with soundness, unlinkability is not a trace property. Soundness asks whether a satisfaction can be found in a trace. In contrast, unlinkability asks whether the adversary can distinguish between two possible worlds. In the context of FAMB, unlinkability means that the adversary should not be able to tell which honest user generated a pseudonym. Since the pseudonym is composed of a signature and fresh randomness, we model the pseudonym by treating the signature as the pseudonym needed to be distinguished by the adversary. So we consider two worlds. In the left world, the signature is generated by user *id*. In the right world, the pseudonym is generated by user *id*1. The adversary should have an identical view of both worlds.

This is exactly what diff mode is designed for. In the rule `sig_diff`, we use the diff operator to output a different signature in the two worlds. In the left world, the output is a signature generated by the first user. In the right world, the output is a signature generated by another user. The rest of the model is mostly the same as the soundness model from the previous chapter.

5.4.1 Model 1

In the first model, we use Tamarin's natural number built-in to model protocol time. This is the most intuitive first choice, as protocol time is a natural number that grows larger. The main challenge rule is shown below.

```
rule sig_diff:
    let
    sig = sign(~m, lsk_signed)
    sig1 = sign(~m1, lsk1_signed)
    in
    [
    In(%t), // time of signing
    //user 1
    Fr(~m),
    In(id),
    !User(id, lsk, lpk, %uct),
    !Ratchet(id, %t, lsk_signed, lpk_signed),
    //user 2
    Fr(~m1),
```

```
    In(id1),
    !User(id1, lsk1, lpk1, %uct1),
    !Ratchet(id1, %t, lsk1_signed, lpk1_signed)
    ]--[OnlyOnce(), SignedTime(%t, %uct, %uct1,id,id1)]->
    [Out(diff(sig, sig))]
```

The premise `In(%t)` lets the adversary choose the protocol time at which the pseudonym is generated. The premises `In(id)` and `In(id1)` let the adversary choose the two challenge users. This follows the usual structure of an indistinguishability game, where the adversary is allowed to choose the two users that it wants to distinguish.

For the first user, the rule requires a user state

```
!User(id, lsk, lpk, %uct)
```

and a ratchet fact for the key

```
!Ratchet(id, %t, lsk_signed, lpk_signed).
```

Similarly, for the second user, the rule requires

```
!User(id1, lsk1, lpk1, %uct1)
```

and

```

!Ratchet(id1, %t, lsk1_signed, lpk1_signed).
```

The tag `'honestok'` indicates that the challenge users should be honest in order to prevent trivial attack (‘`ok`’ stands for “own key”). The ratchet facts provide the signing keys used at the challenge time `%t`. As in the soundness model, the ratchet fact allows the model to refer to a user’s key at a specific protocol time. This make it possible for the challenge signature to be generated after performing ratcheting. The action fact `OnlyOnce()` marks that the challenge rule has fired. Together with the corresponding restriction, it ensures that the challenge is created at most once. The action fact

$$SignedTime(\%t, \%uct, \%uct1, id, id1)$$

is used by restrictions to specify which challenge users are valid to be chosen. In particular, the two challenge users should not be corrupted at or before the challenge signing time. Otherwise, the adversary may already know the identity keys of the challenge users and can distinguish the two signatures trivially. To rule out such traces, we add the following two restrictions:

```
restriction corrupt_after_chall_1:
"All %corrupt_time %uct1 %uct2 %signtime id1 id2 #i #j.
    SignedTime(%signtime, %uct1, %uct2, id1, id2) @ #i
  & CorruptPK(id1,%corrupt_time) @ #j
  ==> %signtime << %corrupt_time"
restriction corrupt_after_chall_2:
"All %corrupt_time %uct1 %uct2 %signtime id1 id2 #i #j.
    SignedTime(%signtime, %uct1, %uct2, id1, id2) @ #i
  & CorruptPK(id2,%corrupt_time) @ #j
  ==> %signtime << %corrupt_time"
```

The first restriction applies to the first challenge user $id1$. It says that if a challenge signing event involving $id1$ and $id2$ occurs at protocol time $\%signtime$, and if $id1$ is corrupted at protocol time $\%corrupt_time$, then the challenge signing time must be strictly earlier than the corruption time:

$$\%signtime << \%corrupt_time.$$

Here << is the strict ordering on natural-number protocol time. The second restriction is symmetric for the second challenge user $id2$.

The final output $Out(diff(sig, sig1))$ means that the adversary receives sig in the left world and $sig1$ in he right world. Therefore, the left world corresponds to the case of user id, and the right world corresponds to the case of $id1$. If Tamarin can prove observational equivalence for this model, then it means that the adversary cannot distinguish which user generated the pseudonym. This is exactly the unlinkability property we want to capture.

5.4.2 Model 2

When trying to prove observational equivalence for Model 1, we found an unreasonable attack. The attack appear to distinguish the two worlds through the protocol time. We suspected that the natural number built-in might interact unexpectedly with diff mode. To test this, we built Model 2. Instead of using natural numbers, we modeled protocol time using the `multiset` built-in. In this representation, time is modeled as a multiset term. A later time is represented by adding a step to the current time $wut = t + stept$. Intuitively, a larger multiset represents a later protocol time. However, this change did not remove the problem. Tamarin still produced an unreasonable attack related to protocol time.

5.4.3 Issue in Diff mode

Through our efforts of separating the issue, it turns out that the problem arises from the interaction between an adversary-controlled input and restrictions defined over trace events.

The following minimalist example illustrates the issue:

```
rule Step:
    [In(t), In(nextT)]
    --[GT(t,nextT)]->
    []

restriction Strict:
    "All #i t1 t2. GT(t1,t2) @ i ==> not (t1 = t2)"

rule sig_diff:
    [In(id1)
    ]--[Diff(id1)]->
    [Out(diff(id1, id1))]
```

Listing 5.1: Minimal Tamarin model illustrating the bug

The rule `Step` allows the adversary to provide two inputs t and $nextT$. The restriction `Strict` enforces that the chosen inputs must be unequal. Running the prover in diff mode, Tamarin reports an "attack trace". We observed that removing either the rule or the restriction eliminates the false attack. We reported this issue to the Tamarin developers and opened a GitHub issue [18]. The official feedback for now was to avoid using restrictions in diff mode when possible.

Chapter 6

Unlinkability in Other Models

In this chapter, we give an overview from all official examples of Tamarin that use diff mode to successfully prove similar unlinkability properties for protocols of comparable scale. This chapter aims to give some ideas of why the unlinkability of FAMB explodes Tamarin.

6.1 5G-AKA Untraceability

In 2018, Basin et al. gives a formal analysis of the next generation mobile communication key agreement protocol (5G-AKA) using Tamarin [4]. 5G-AKA model contains three main parties: the subscriber, also called the user equipment (UE), the serving network (SEAF), and the home network (HSS). The UE and HSS share a long-term symmetric key K, a permanent subscriber identity SUPI, and sequence-number state. The SEAF does not know the subscriber's long-term key. Its role is to forward messages between the UE and HSS and to store the authentication material returned by the HSS. Merely preventing the attackers from learning identifying data (i.e., SUPI) does not prevent traceability attacks, which distinguish subscribers and can be exploited to track subscribers over time. Thus, the untraceability property of 5G-AKA was evaluated in Tamarin. The authors considered both active adversaries and passive adversaries. An active attacker can set up fake base stations to send and receive signaling messages. And a passive attacker can only eavesdrop on all messages but cannot emit a signal.

For the active-adversary setting in the Tamarin file "5G_AKA_priv.spthy", the authors provide a full Tamarin model of 5G-AKA. This model is organized into several blocks. The first part declares the theory, the built-in equational theories, the AKA functions, and

several tactics for reasoning about sequence numbers. The model then uses 2 rules to set up the secure channel between SEAF and HSS. Next, it contains 4 rules for the initialization section. It initializes the serving network, initializes the home network, generates the two SUPIs needed to be distinguished, and initializes the subscription between a subscriber and its home network. This subscription state contains the long-term symmetric key and the sequence-number state for both UE and HSS. The main protocol rules form the largest part of the model, using 11 rules and 275 lines of code. This part models the full 5G-AKA message flow. First, the UE sends an attach request containing the concealed subscriber identity SUCI. The SEAF then forwards this request to the HSS as an authentication initiation request (AIR). The HSS processes the AIR, decrypts the SUCI, looks up the subscription state, and constructs the 5G authentication vector. The SEAF receives this authentication vector and sends the authentication request to the UE. The UE then has two possible branches: if the MAC and freshness checks succeed, it computes and sends the authentication response; if the MAC check succeeds but the freshness check fails, it sends a synchronization failure message. The SEAF either forwards the authentication response to the HSS as a confirmation message, or forwards the synchronization failure. Finally, the HSS either confirms the authentication and the SEAF completes the protocol, or the HSS processes the resynchronization message and updates its sequence number. In total, this file uses around 490 lines of code to model the protocol interactions. The authors then manually guide Tamarin to find an attack trace that is able to distinguish the two SUPIs.

For the passive-adversary setting, they prove untraceability instead, but only for a heavily simplified model. The model is much smaller than the full active-adversary model. The file contains only around 101 LOC in total for modeling protocol behaviors. Same as the active model, the first part declares the theory, the built-ins, the AKA functions, and a tactic for observational equivalence. The main modeling part reuses the same initialization section as the active model, with the same 4 initialization rules. In contrast to the full model specified in the active adversary setting, this file does not model the complete message-by-message interaction between UE, SEAF, and HSS. Instead, it keeps only the information needed for the passive untraceability game. The only protocol behavior is then collapsed into the rule `session`. This rule does not separately model the attach request, authentication initiation request, authentication vector generation, authentication request, authentication response, confirmation, or resynchronization steps. Instead, it directly produces the observable values of one 5G-AKA session: the concealed identity `suci`, the pair containing `RAND` and `AUTN`, and the derived response value `RES_star`. The sensitive values, including the anonymity key, the long-term key, the sequence number, and the SUPI, are marked with `Secret` action facts. Even in this reduced setting, the proof files contain 107,509 lines [16].

6.2 ECC-DAA Unlinkability

In 2020, Wesemeyer et al. gives a formal analysis of a TPM 2.0-based Direct Anonymous Attestation scheme (DAA) [19]. DAA is a special group signature scheme where the group manager (Issuer) issues credentials which can be used to verify that a signature was created by a valid member of the group without identifying the signer. In their setting, a platform is the combination of a Host and a TPM, where the TPM is the trusted hardware chip that provides cryptographic primitives and secure storage.The scheme has five main operations: Setup, Join, Sign, Verify, and Link. Setup publishes the system parameters and the issuer's public keys. Join lets a Host with a TPM obtain an attestation credential for an Ecc-DAA key and joins the group. Sign is then used to anonymously sign a message, attest keys, or certify a value. Verify checks whether a signature is valid. Link checks whether two signatures can be linked. In particular, signatures produced with the same Ecc-DAA key and the same basename can be linked, while signatures with different basenames should not reveal whether they come from the same platform.

The unlinkability property considered in this example is called user-controlled unlinkability. Informally, it says that, given two messages associated with different basenames, an adversary who does not know the corresponding platform private keys should not be able to tell whether the two messages were produced by the same platform or not.

The authors define seven security properties, SP1–SP7, to be verified in Tamarin. These properties include correctness, user-controlled linkability, unforgeability, non-frameability, anonymity, user-controlled unlinkability, and agreement during the Join phase. Because the full ECC-DAA protocol is too large to verify as one single model, they split the analysis into three parts: Join–Sign–Verify, Join–Certify–Verify, and Join–Quote–Verify. The Sign, Certify, and Quote cases correspond to the three signing contexts supported by their TPM 2.0-based DAA scheme.

For each part, the authors further use 2 different model variants. To prove the trace properties, such as correctness, linkability, unforgeability, non-frameability, and agreement, they used a complete modeling of the protocol that kept all the fine-grained protocol interactions. They explicitly modeled the secure channel between the Host and the TPM, the platform setup, the Join protocol, and all the interactions between the TPM, Host, Issuer, and Verifier. The Join–Sign–Verify model alone requires almost 1400 lines of Tamarin code and more than 38 rules, and that the interaction between the TPM, Host, Issuer, and Verifier accounts for another about 1000 lines of code. However, the anonymity and user-controlled unlinkability properties are proved using observational equivalence in a simplified version of the protocol. Specifically, they only initialize the values that are needed

to construct the two objects compared in the indistinguishability experiment. The reduced model treats Host and TPM as a single entity, and omit all interactions of the protocol. Take the Sign case for example, the reduced model contains only 3 rules: the first two rules initialize the issuer, and the main rule directly generates the signatures to be distinguished. Even with this simplification, the generated proof files are still very large: the unlinkability proof file contains 71,244 lines for the Sign part, 106,984 lines for the Certify part, and 80,089 lines for the Quote part [17].

6.3 Others

There are two other examples that are related to our setting: Dispute Resolution in Voting [5] and RFID [8]. However, compared with 5G-AKA, DAA, and FAMB, these protocols have much lower modeling complexity. They involve fewer parties, fewer stateful components, and fewer protocol interactions. In the Dispute Resolution in Voting example, the protocol only contains two entities: one authority and one voter. The Tamarin analysis decomposes the proof by separating the different honest and dishonest cases into different files. Thus, instead of proving one large model covering all combinations of behavior, the analysis splits the cases and proves them separately. The RFID example follows a similar methodology. The protocol itself is much smaller, with interactions mainly between a reader and a tag. The Tamarin models again separate different cases of the reader and tag into different files, and each case is proved independently.

6.4 Discussion

The examples in this chapter show a common pattern. Tamarin can only prove unlinkability-style properties using diff mode using a collapsed model. In 5G-AKA, the active model keeps the full protocol interaction and leads to an attack trace, while the passive untraceability proof is done on a much more reduced model whereas the whole interactions collapsed into one rule that generates all final values. In ECC-DAA, the trace properties are proved on fine-grained models with many protocol interactions, but the unlinkability proof is done on simplified models that collapse the protocol and directly generate the objects being distinguished. Smaller examples such as Voting and RFID are also decomposed into separate cases and proved file by file.

However, these two approaches are not very suitable for FAMB. First, if we follow the strategy used in the reduced 5G-AKA and ECC-DAA models and remove most protocol

interactions, then the resulting model would only generate two signatures and compare them. This would no longer capture the source of linkability in FAMB. In FAMB, linkability is most likely to arise from stateful protocol behavior, such as corruption, ticking, ratcheting, and follow relations. If these interactions are omitted, then the proof only says that two fresh signatures are indistinguishable in isolation. This statement is too weak and is almost independent of the FAMB protocol itself.

Second, decomposing the proof into many small cases is also not fully satisfactory. The unlinkability property of FAMB depends on many combinations of state: whether a user has been corrupted, when the corruption happens, whether a follow relation exists, whether the follower's stored key has been ratcheted, and what protocol time is used for the challenge. Splitting the model into separate cases may make individual proofs easier, but it is difficult to argue convincingly that all relevant combinations have been covered. Missing one case could remove exactly the trace where linkability appears.

This suggests an important modeling lesson. Diff mode has a much larger search space than ordinary trace-property proving, because Tamarin has to reason about two systems at the same time and mirror each step between them. If a protocol contains many rounds of interaction, more than two entities, and many adversary-controlled rules, then the proof can easily become intractable. This is especially true when the model contains state-changing rules such as `Tick`, `Trust`, corruption, or ratcheting rules. These rules can often be applied in many different orders, and create different loops in the proof search.

Therefore, for protocols of this kind, there are two practical approaches. The first approach is to collapse the model into a much simpler privacy experiment, where only the values needed for the indistinguishability challenge are generated. The second approach is to decompose the protocol into smaller cases and prove each case separately. If neither of these reductions is possible, then Tamarin's diff mode is not a good direct tool for proving unlinkability on the full model. In such cases, it is better to dive into pen-and-paper proof.

Chapter 7

Conclusion

The practical motivation for this work is illustrated by Albrecht et al.'s study of protesters [2]. They found that participants relied on pseudonymity and developed collective practices for responding to arrests and compromised devices to seek forward anonymity. These findings demonstrate both the value and the difficulty of decentralized messaging during internet shutdowns. In our view, Anix [11] is a promising design for this setting because it addresses the tension between anonymity and trust through selective recognition, remote trust establishment, and anonymous message endorsement, while operating in a mesh network.

This thesis studied the security of Anix. Although Anix generates a fresh pseudonym for every broadcast, revealing a user's long-term public identity key may allow an adversary to recognize pseudonyms generated in earlier broadcasting. To address this problem, we proposed a key-ratcheting component that evolves both the user's signing key and the corresponding public identity key. Under the random-oracle model, we gave a correctness argument showing that the ratcheted public and secret keys remain consistent, and a one-wayness argument showing that a later key should not reveal its predecessors. This provides forward anonymity, ensuring that compromise of a user's current public identity key does not allow the adversary to identify messages previously sent by that user.

We then defined the resulting protocol as a forward-anonymous message broadcast system, called FAMB. The formalization specifies the protocol algorithms, user state, follow relationships, corruption behavior, protocol time, and the adversarial oracles used in the security games. We then gave an instantiation of FAMB using Ed25519 and HPKE.

We developed a symbolic Tamarin model of FAMB and proved its soundness as an unforgeability property. The proof shows that a signature verifying under an honestly

generated public key must either have been honestly produced earlier or have been preceded by disclosure of the corresponding signing key. We also repeated the analysis using the weaker SVS model of signatures, which admits a broader range of adversarial behavior in signature verification. Tamarin was still able to prove the soundness property under this weaker model.

For unlinkability, we attempted to prove observational equivalence using Tamarin's diff mode. We constructed models using both the natural-number and multiset representations of protocol time. During this analysis, we identified an issue in Tamarin diff mode involving the interaction between adversary-controlled inputs and restrictions. The issue produces an apparent distinguishing trace even in a small example where no meaningful distinction should exist. We reported this behavior to the Tamarin developers. Avoiding the problematic restrictions leads instead to a proof search that becomes too large to complete for the full FAMB model.

To better understand this limitation, we reviewed existing Tamarin analyses of unlinkability and related privacy properties, including 5G-AKA, ECC-DAA, voting, and RFID examples. A common pattern is that successful equivalence proofs either collapse the full protocol into a small number of rules that directly generate the challenge objects, or divide the analysis into many separate cases. Neither approach is fully satisfactory for FAMB. Removing corruption, trust establishment, ratcheting, and time evolution would remove the main sources from which linkability may arise, while dividing the protocol into many cases makes it difficult to argue that all relevant combinations of state have been covered.

The main direction for future work is therefore to give a complete pen-and-paper proof of unlinkability for the Anix-based instantiation of FAMB. At the beginning of this thesis, one motivation for using Tamarin was to avoid the lengthy and potentially error-prone case analysis required by such a proof. However, since the observational-equivalence analysis did not establish unlinkability, a manual reduction thus needed to provide confidence in the full construction. Such a proof should reduce any successful linking adversary to an adversary against the one-wayness of the ratchet, while explicitly accounting for trust establishment, adaptive corruption, key evolution, and out-of-sync users. Future improvements to Tamarin's equivalence reasoning may make it possible to reproduce this result automatically. Overall, the results suggest that Tamarin is effective for trace properties such as soundness, but its diff mode should be used with care for protocols containing many state-changing and adversary-controlled operations.

References

[1] Martín Abadi and Phillip Rogaway. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)*. *Journal of Cryptology*, 15(2):103–127, January 2002.

[2] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: The case of Hong Kong. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3363–3380. USENIX Association, August 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/albrecht.

[3] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022. https://www.rfc-editor.org/rfc/rfc9180.

[4] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1383–1396, Toronto Canada, October 2018. ACM.

[5] David Basin, Saša Radomirović, and Lara Schmid. Dispute Resolution in Voting. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 1–16, June 2020. ISSN: 2374-8303.

[6] Frank Denis, Edward Eaton, Tancrède Lepoint, and Christopher A. Wood. Key Blinding for Signature Schemes. Internet Draft draft-irtf-cfrg-signature-key-blinding-06, Internet Engineering Task Force. Num Pages: 16.

[7] Danny Dolev and Andrew Chi-Chih Yao. On the Security of Public Key Protocols. *IEEE Trans. Inf. Theor.*, 29(2):198–208, 1983.

[8] Jannik Dreier, Lucca Hirschi, Saša Radomirović, and Ralf Sasse. Verification of stateful cryptographic protocols with exclusive OR. *Journal of Computer Security*, 28(1):1–34, February 2020.

[9] Edward Eaton, Tancrède Lepoint, and Christopher A. Wood. Security analysis of signature schemes with key blinding. Cryptology ePrint Archive, Paper 2023/380, 2023. Preprint.

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

[11] Sina Kamali and Diogo Barradas. Anix: Anonymous blackout-resistant microblogging with message endorsing. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1381–1399. IEEE, May 2025.

[12] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 696–701. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science.

[13] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[14] Signal. The Double Ratchet Algorithm, 2025. /web.archive.org/web/20260714202417/https://signal.org/docs/specifications/doubleratchet/.

[15] Gert Smolka, Werner Nutt, Joseph A. Goguen, and José Meseguer. Order-Sorted Equational Computation. In *Rewriting Techniques*, pages 297–367. Elsevier, 1989.

[16] Tamarin Prover Developers. 5G-AKA Tamarin Models, 2018. /web/20260714202948/https://github.com/tamarin-prover/tamarin-prover/tree/develop/examples/ccs18-5G/5G-AKA-untraceability.

[17] Tamarin Prover Developers. Tamarin eccDAA, 2018. /web/20260714202717/https://github.com/tamarin-prover/tamarin-prover/tree/develop/examples/asiaccs20-eccDAA.

[18] Tamarin Prover Developers. Tamarin Issue, 2018. https://web.archive.org/web/20260714201924/https://github.com/tamarin-prover/tamarin-prover/issues/815.

[19] Stephan Wesemeyer, Christopher J.P. Newton, Helen Treharne, Liqun Chen, Ralf Sasse, and Jorden Whitefield. Formal Analysis and Implementation of a TPM 2.0-based Direct Anonymous Attestation Scheme. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 784–798, Taipei Taiwan, October 2020. ACM.