PRIVACY-PRESERVING ACCOUNTABILITY IN ONLINE MESSAGING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Nirvan Tyagi

August 2023

© 2023 Nirvan Tyagi

ALL RIGHTS RESERVED

ABSTRACT

Technologies that enable confidential communication and anonymous authentication are important for providing privacy for users of internet services. Unfortunately, encryption and anonymity, while good for privacy, make it hard to hold bad actors accountable for misbehavior. Internet services rely on seeing message content to detect spam and other harmful content; services must also be able to identify users to attribute and respond to abuse complaints. This tension between privacy and accountability leads to one of two suboptimal outcomes: Services require excessive trust in centralized entities to hold users accountable for misbehavior, or services leave themselves and/or their users open to abuse.

In this dissertation, I will examine where this tension arises in our modern private messaging systems and how gaps in accountability can and do lead to real-world attacks. I will discuss how I have addressed this tension through the design of new cryptographic protocols. In particular, I will present new protocols for secure abuse-reporting, anonymous blocklisting, and transparent key infrastructure.

BIOGRAPHICAL SKETCH

Nirvan Tyagi was born and raised in Ames, Iowa where he attended Ames High School. He completed his undergraduate studies at Massachusetts Institute of Technology, majoring in Computer Science. After participating in a variety of undergraduate research opportunities, he discovered an affinity for building secure systems in the Parallel and Distributed Operating Systems (PDOS) Group at MIT where he stuck around to receive a Master of Engineering degree. Continuing his research in this area, he joined the Security Group at Cornell University to pursue his graduate studies. After spending one year on the Ithaca campus, he moved to New York City to join the first cohort of students on the new Cornell Tech campus on Roosevelt Island. Now having completed his doctoral degree, he will spend a year as a postdoctoral scholar with the Secure Computer Systems Group at Stanford University, and then he will join the faculty as an assistant professor at University of Washington.

To my parents, my Baba, my Dadu. I am honored to follow in your footsteps.

ACKNOWLEDGEMENTS

None of the work in this thesis would have been possible without the superb guidance of my advisor, Tom. I am lucky to have had his support and advice through every turn of my graduate journey. One of the most enjoyable aspects of my journey was getting to work with a huge set of amazing people across Cornell, academia, industry, and government, including two wonderful research visits at University of Washington and Stanford. Again, I attribute this highly collaborative and interdisciplinary experience to my advisor Tom who has a knack for bringing people together.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vi

1 Introduction 1

2 Preliminaries 3
2.1 Pseudocode Security Games . . . 3
2.2 Groups and Hardness Assumptions . . . 4
2.3 Idealized Models . . . 7
2.4 Cryptographic Primitives . . . 7

3 Secure Reporting for Content Moderation 14
3.1 Asymmetric Message Franking . . . 15
3.1.1 Meaningful Deniability in Messaging . . . 17
3.1.2 Syntax and Security Notions . . . 18
3.1.3 Construction . . . 24
3.1.4 Security Analysis . . . 28
3.2 Message Tracing . . . 31
3.2.1 Traceback Setting in Messaging . . . 33
3.2.2 Syntax and Security Notions . . . 36
3.2.3 Construction . . . 44
3.2.4 Security Analysis . . . 49
3.2.5 Evaluation . . . 54
3.3 Follow-up Work . . . 57

4 Sender-Anonymous Blocklisting 59
4.1 Sender Anonymity in Messaging . . . 62
4.1.1 Background: Signal and Sealed Sender . . . 63
4.1.2 Limitations of Sealed Sender . . . 65
4.2 Outsourced Blocklisting . . . 67
4.3 Blocklisting from Group Signature . . . 72
4.3.1 Group Signature Syntax and Security . . . 72
4.3.2 Construction of Group Signature . . . 79
4.3.3 Security Analysis . . . 82
4.3.4 Outsourced Blocklisting from Group Signatures . . . 87
4.4 Extending Blocklisting with One-time Use Tokens . . . 88
4.5 Evaluation . . . 93

5 Verifiable Public Key Infrastructure 97
5.1 Auditing Public Key Infrastructure in Messaging . . . 103
5.2 Versioned Invariant Proofs for RSA Authenticated Dictionaries . . . 107
5.2.1 RSA Authenticated Dictionary . . . 109
5.2.2 Versioned Invariant Update Proofs and Strong Key Binding . . . 111
5.3 Authenticated History Dictionaries . . . 115
5.3.1 Syntax and Security Notions . . . 115

5.3.2 AHD Constructions 118
5.3.3 Security Analysis 123
5.4 Client Checkpoint Auditing 125
5.5 Evaluation 134
5.5.1 Implementation 134
5.5.2 Client Auditing Costs 136
5.5.3 Server Epoch Update Costs 137
5.5.4 Key Lookup Costs 140

6 Ethics Discussion **142**

7 Concluding Thoughts **147**

Bibliography **148**

CHAPTER 1

INTRODUCTION

Technologies that enable confidential communication and anonymous authentication are important for preserving privacy for users communicating over the internet. They work by shifting control of cryptographic keys that secure communication and authentication directly to user devices. Unfortunately, encryption and anonymity, while good for privacy, make it hard to hold bad actors accountable for misbehavior. Internet services rely on reading messages to detect spam and harmful content; services must also be able to identify users to attribute and respond to abuse complaints. This tension between privacy and accountability leads to one of two suboptimal outcomes: Services require excessive trust in centralized entities to hold users accountable for misbehavior, or services leave themselves and/or their users open to abuse.

Some examples of where this tension arises in modern messaging platforms for private communication include:

- On WhatsApp, organized disinformation campaigns have contributed to political unrest and social problems. State-of-the-art moderation approaches for fighting viral disinformation in plaintext settings rely on identifying the source account of a message—information that is shielded by end-to-end encrypted forwarding.
- In a sender-anonymous messaging protocol deployed by Signal, I identified attacks leaving recipients open to spam from anonymous senders. Outsourcing blocklisting of senders and spam to the platform or even reporting abusive senders to the platform moderator, as is usually done on messaging platforms, is not possible when the sender identity is hidden.
- On centralized services like WhatsApp and Signal, users must trust the service not to undermine end-to-end encryption through corrupt identity management. This represents a problematic trust relationship as the services providing the identity infrastructure are the same services that may have incentive to access private communications.

I have addressed the above problems by introducing new abuse-resistant protocols for secure message traceback [TMR19a], sender-anonymous abuse reporting [TGL+19a] and blocklisting [TLMR22], and transparent public key infrastructure [TFZ+22]. A unifying theme among all these works is a methodological approach where I work backwards from real problems and develop

new cryptographic theory to guide sound protocol design. I implement and evaluate my proposed cryptographic protocols within real systems against performance constraints that exist in practice.

In this dissertation, we will step through this methodology in each chapter applying it to build new systems to facilitate private communication while holding bad actors accountable.

CHAPTER 2

PRELIMINARIES

2.1 Pseudocode Security Games

We formalize security using the code-based game approach of Bellare and Rogaway [BR06]. We will use a concrete security approach in which we account for adversarial resources explicitly in theorem statements, rather than defining security asymptotically. Asymptotic notions can be derived from our treatment in a straightforward way.

We use $x \leftarrow y$ and $x \leftarrow \mathsf{Eval}()$ to denote assigning the value of y and the evaluation of Eval to variable x. If Eval uses random coins, we instead denote $x \leftarrow\!\!\$\ \mathsf{Eval}$. We denote the fixed input length and output length of algorithms when appropriate as $\mathsf{Eval.il}$ and $\mathsf{Eval.ol}$, respectively. For finite set Y, we denote $x \leftarrow\!\!\$\ Y$ as sampling a random value from the set. We denote a dictionary D initialized as $[\cdot]$ to store key-value pairs (k,v). Adding or updating a value v for key k is denoted as $D[k] \leftarrow v$. A table T is a special use of a dictionary in which values are added in sequence with incrementing keys. We denote appending a value v to a table with $T \hookleftarrow v$. We will allow for membership queries on dictionaries of the form $k \in D$, $v \in D$, and $(k,v) \in D$, also allowing for wildcard queries of the form $(k,*) \in D$.

Interactive protcols. To model interactive protocols between two parties, following the treatment of previous work [BMW03, BSZ05], we define an algorithm for each party that takes an incoming message and a current state, and returns an outgoing message, an updated state, and a decision in $\{\texttt{accept}, \texttt{reject}, \texttt{cont}\}$. If the decision is `accept`, the output of the protocol for the party will be stored in the state.

Compact ranges. A *compact range* is a succinct, canonical representation of a range $[L,R)$ where L,R are non-negative integers [MKL+20]. A compact range, $[(L_i,R_i)]_{i=1}^{m} \leftarrow \mathsf{CompactR}((L,R))$, is the minimum set of m subranges that "span" $[L,R)$ where $L_1 = L$, $R_m = R$, and $R_i = L_{i+1}$ for all $1 \leq i < m$. Each subrange is restricted to be of the form: $(L_i = a_i \cdot 2^{b_i}, R_i = L_i + 2^{b_i})$ for non-negative integers (a_i, b_i). It is guaranteed that a unique compact range exists for every range; further, the time to compute the compact range and the number of subranges m is logarithmic in the size of the range, $\mathcal{O}(\log(R-L))$.

2.2 Groups and Hardness Assumptions

Prime-order cyclic groups. We will make use of prime-order cyclic groups for which we will notate as $\mathbb{G}$ with prime order p. We assume an efficient setup algorithm GGen that on input security parameter λ, generates a group, $(p, \mathbb{G}, g) \leftarrow \mathsf{GGen}(\lambda)$, where $|p| = \lambda$.

Bilinear pairing groups. Our constructions will make use of bilinear pairing groups for which we will use the following notation. (1) Groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of prime order p. (2) Group element g_1 is a generator of $\mathbb{G}_1$, g_2 is a generator of $\mathbb{G}_2$. (3) Pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is a computable map with the following properties: *Bilinearity*: $\forall\ u \in \mathbb{G}_1, v \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$, and *Non-degeneracy*: $e(g_1, g_2) \neq 1$. We assume an efficient setup algorithm BGGen that on input security parameter λ, generates a bilinear group, $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e) \leftarrow \mathsf{BGGen}(\lambda)$, where $|p| = \lambda$.

RSA groups. An *RSA group* is the multiplicative group of invertible integers modulo N (denoted $\mathbb{Z}_N^\times$), where N is the product of two secret primes. We define the *RSA quotient group* for N as $\mathbb{Z}_N^\times \setminus \{\pm 1\}$. The widely believed Strong RSA Assumption (Strong-RSA) asserts that it is computationally difficult to compute e^{th} roots of a non-trivial element of $\mathbb{Z}_N^\times$ for $e \geq 3$.

<u>*Groups of unknown order.*</u> We assume the existence of a randomized polynomial time sampling algorithm $\mathsf{HGGen}(\lambda)$ that takes as input the security parameter λ and generates a group of unknown order consisting of two integers a, b along with a description of the group $\mathbb{G}$. The group $\mathbb{G}$ is of unknown order in the range $[a, b]$ where a, b, and $a - b$ are all exponential in λ.

The RSA quotient group $\mathbb{Z}_N^\times \setminus \{\pm 1\}$ where N is an RSA modulus is believed to have no element of known order other than the identity. The group generation algorithm here may require trusted setup to generate the group modulus N.

<u>*Extended Euclidean algorithm.*</u> Given two integers x, y such that the $\gcd(x, y) = 1$, then $(a, b) \leftarrow \mathsf{EEA}(x, y)$ returns the Bézout coefficients (a, b) where $ax + by = 1$. The coefficients are such that $a \leq y$ and $b \leq x$. The algorithm runs in time $\mathcal{O}(\max(|x|, |y|))$.

Discrete logarithm (DL). The discrete log assumption is defined by the security game $\mathrm{DL}_{\mathsf{GGen}}^{\mathcal{A}}(\lambda)$ in

which an adversary is tasked with finding the discrete log of a random group element. The advantage of an adversary is defined as $\mathbf{Adv}^{\mathrm{dl}}_{\mathsf{GGen},\mathcal{A}}(\lambda) = \Pr[\mathrm{DL}^{\mathcal{A}}_{\mathsf{GGen}}(\lambda) = 1]$. We will also make use of the discrete log assumption in $\mathbb{G}_1$ and $\mathbb{G}_2$ of the bilinear pairing groups, which is one of the assumptions made by external Diffie-Hellman (XDH).

Decisional Diffie-Hellman assumption (DDH). The decisional Diffie-Hellman (DDH) assumption is defined by the security game $\mathrm{DDH}^{\mathcal{A},b}_{\mathsf{GGen}}(\lambda)$ in which an adversary is tasked with distinguishing between a triple of random group elements and a random Diffie-Hellman triple. The advantage of an adversary is defined as $\mathbf{Adv}^{\mathrm{ddh}}_{\mathsf{GGen},\mathcal{A}}(\lambda) = \left|\Pr[\mathrm{DDH}^{\mathcal{A},1}_{\mathsf{GGen}}(\lambda) = 1] - \mathrm{DDH}^{\mathcal{A},0}_{\mathsf{GGen}}(\lambda) = 1]\right|$. Again, we will make use of the DDH assumption in $\mathbb{G}_1$ and $\mathbb{G}_2$ of the bilinear pairing groups, which is one of the assumptions made by external Diffie-Hellman (XDH).

Decision linear assumption (DLIN). The decision linear (DLIN) assumption is defined by the security game $\mathrm{DLIN}^{\mathcal{A},b}_{\mathsf{GGen}}(\lambda)$ in which an adversary is tasked with distinguishing between a set of three random group elements along with those same three values taken to different random exponents and a set where the last group element is not taken to a random exponent but the sum of the previous two exponents. The decision linear assumption is considered to hold even in groups where DDH is easy and thus is thought to hold in pairing groups even when the associated group elements in the paired group are revealed. In our version of the game, we explicitly return the group elements in $\mathbb{G}_2$ since we will need to make use of them in our reductions. If we used a pairing type with an efficiently computable isomorphism then we would not need to this change. This variant is sometimes referred to as the external decision linear assumption (XDLIN) [LPY15]. The advantage of an adversary is defined as $\mathbf{Adv}^{\mathrm{dlin}}_{\mathsf{BGGen},\mathcal{A}}(\lambda) = \left|\Pr[\mathrm{DLIN}^{\mathcal{A},1}_{\mathsf{BGGen}}(\lambda) = 1] - \mathrm{DLIN}^{\mathcal{A},0}_{\mathsf{BGGen}}(\lambda) = 1]\right|$.

Knowledge of exponent (KEA). The knowledge of exponent assumption [Dam91] concerns triples of the form (g^a, g^b, g^{ab}) (i.e. a Diffie-Hellman (DH) triple) for a prime-order group $\mathbb{G}$ of order p (with p prime) and generator g. It says, roughly, that an adversary which on input g^a outputs a DH triple must "have knowledge" of the exponent b which can be extracted from its description. Formally, for an adversary $\mathcal{A}$ and extractor $\mathsf{X}_{\mathcal{A}}$ (which, crucially, is relative to $\mathcal{A}$), we define the *knowledge of*

Game $\mathrm{DL}^{\mathcal{A}}_{\mathsf{GGen}}(\lambda)$
$(p, \mathbb{G}, g) \leftarrow \mathsf{GGen}(\lambda)$
$x \leftarrow \mathbb{Z}_p$
$x' \leftarrow\$ \mathcal{A}(g^x, g)$
Return $x = x'$

Game $\mathrm{DDH}^{\mathcal{A},b}_{\mathsf{GGen}}(\lambda)$
$(p, \mathbb{G}, g) \leftarrow \mathsf{GGen}(\lambda)$
$(\alpha, \beta, \gamma) \leftarrow\$ \mathbb{Z}_p$
$C_0 \leftarrow g^\gamma$; $C_1 \leftarrow g^{\alpha\beta}$
$b' \leftarrow\$ \mathcal{A}(g^\alpha, g^\beta, C_b, g)$
Return b'

Game $\mathrm{DLIN}^{\mathcal{A},b}_{\mathsf{BGGen}}(\lambda)$
$(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e) \leftarrow \mathsf{BGGen}(\lambda)$
$(\alpha, \beta, \gamma) \leftarrow\$ \mathbb{Z}_p$
$(m, n, l) \leftarrow\$ \mathbb{Z}_p$
$m_1 \leftarrow g_1^m$; $m_2 \leftarrow g_2^m$
$n_1 \leftarrow g_1^n$; $n_2 \leftarrow g_2^n$
$l_1 \leftarrow g_1^l$; $l_2 \leftarrow g_2^l$
$C_0 \leftarrow l_1^\gamma$; $C_1 \leftarrow l_1^{\alpha+\beta}$
$b' \leftarrow\$ \mathcal{A}(m_1, n_1, l_1, m_1^\alpha, n_1^\beta, C_b, m_2, n_2, l_2)$
Return b'

Game $\mathrm{KEA}^{\mathcal{A}}_{\mathsf{GGen},\mathsf{X}_\mathcal{A}}(\lambda)$
$(p, \mathbb{G}, g) \leftarrow \mathsf{GGen}(\lambda)$
$x \leftarrow \mathbb{Z}_p$
$(Y, C) \leftarrow\$ \mathcal{A}(g^x, g)$
$c \leftarrow\$ \mathsf{X}_\mathcal{A}(g^x, g)$
Return $Y = C^x \wedge C \neq g^c$

Figure 2.1: Security games for discrete logarithm-based hardness assumptions.

exponent advantage of $\mathcal{A}$ relative to $\mathsf{X}_\mathcal{A}$ as

$$\mathbf{Adv}^{\text{kea}}_{\mathsf{GGen},\mathsf{X}_\mathcal{A},\mathcal{A}}(\lambda) = \Pr[\mathrm{KEA}^{\mathcal{A}}_{\mathsf{GGen},\mathsf{X}_\mathcal{A}}(\lambda) = 1] \,.$$

Pseudocode for the game KEA is in Figure 2.1. Our formalization follows Bellare et al. [BP04] except with randomized Turing machines instead of families of circuits. Our pseudocode does not depict the random tapes used by $\mathcal{A}$ or $\mathsf{X}_\mathcal{A}$, but the extractor is always given the random tape of the adversary.

Since the output of game KEA involves two adversaries (one of which depends on the other) interpreting it is subtle. The KEA game basically measures the probability $\mathsf{X}_\mathcal{A}$ *fails* to extract the exponent when $\mathcal{A}$ outputs a valid DH triple. The negation of the success condition is a disjunction of two events: either $\mathcal{A}$ does not output a DH triple or $\mathsf{X}_\mathcal{A}$ successfully extracts from $\mathcal{A}$.

Strong RSA assumption. The strong RSA assumption tasks an adversary with computing a chosen non-trivial root of a random group element. We define the advantage of an adversary $\mathcal{A}$ against the strong RSA assumption as follows:

$$\mathbf{Adv}^{\text{strong-rsa}}_{\mathsf{HGGen},\mathcal{A}}(\lambda) = \Pr\left[\begin{array}{l} u^\ell = w \\ \ell \in \mathsf{Primes}(\lambda) \setminus \{2\} \end{array} : \begin{array}{l} (a, b, \mathbb{G}) \leftarrow\$ \mathsf{HGGen}(\lambda); \\ w \leftarrow\$ \mathbb{G}; \\ (u, \ell) \leftarrow\$ \mathcal{A}(a, b, \mathbb{G}, w) \end{array}\right].$$

2.3 Idealized Models

Algebraic group model. In some of our security proofs, we consider security against *algebraic* adversaries which we model using the algebraic group model, following the treatment of [FKL18]. We call an algorithm $\mathcal{A}$ *algebraic* if for all group elements Z that are output (either as final output or as input to oracles), $\mathcal{A}$ additionally provides the representation of Z relative to all previously received group elements. The previous received group elements include both original inputs to the algorithm and outputs received from calls to oracles. More specifically, if $[X]_i$ is the list of group elements $[X_0, \ldots, X_n] \in \mathbb{G}$ that $\mathcal{A}$ has received so far, then, when producing group element Z, $\mathcal{A}$ must also provide a list $[z]_i = [z_0, \ldots, z_n]$ such that $Z = \prod_i X_i^{z_i}$.

Random oracle model. Looking ahead, we will prove security in the random oracle model, modeling hash functions as random oracles. In this model, to each definition we add another oracle RO. The adversary $\mathcal{A}$ and scheme algorithms all have access to it as an oracle. The oracle accepts queries on arbitrary length bit strings m and returns a random bit string r of specified length, e.g., 2λ. It stores r in a table T indexed by m to answer future queries consistently. In some security proofs we will use a technique referred to as programming the random oracle (setting certain oracle outputs to values in a way advantageous to a reduction).

2.4 Cryptographic Primitives

Pseudorandom functions. A pseudorandom function PRF is a tuple of algorithms $(\mathsf{Setup},\ \mathsf{Keygen},\mathsf{Ev})$. The setup algorithm produces the public parameters for the scheme, $pp \leftarrow\$ \mathsf{PRF.Setup}(\lambda)$. The key generation algorithm outputs a key, $k \leftarrow\$ \mathsf{PRF.Keygen}^{pp}()$. The evaluation algorithm produces a deterministic evaluation of the function on an input, $y \leftarrow \mathsf{PRF.Ev}^{pp}(k, x)$.

We consider two security properties for pseudorandom functions. The first is pseudorandomness which ensures the PRF acts as a random function when its key remains secret, for which we consider a multi-key variant. The second is collision resistance which means it is hard to find key-input pairs that evaluate to the same output. The properties are defined by games PRF and CR and we define

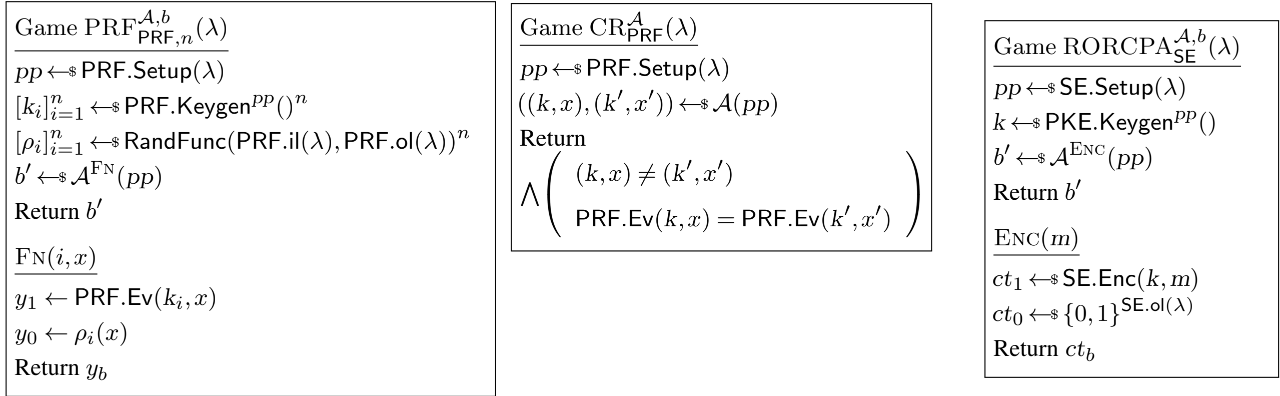

Figure 2.2: Security games for pseudorandom functions and symmetric encryption.

the advantage of an adversary against each of these games as follows:

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{PRF},\mathcal{A},n}(\lambda) = \left|\Pr[\mathrm{PRF}^{\mathcal{A},1}_{\mathsf{PRF},n}(\lambda)=1] - \mathrm{PRF}^{\mathcal{A},0}_{\mathsf{PRF},n}(\lambda)=1]\right|,$$

$$\mathbf{Adv}^{\text{cr}}_{\mathsf{PRF},\mathcal{A}}(\lambda) = \Pr[\mathrm{CR}^{\mathcal{A}}_{\mathsf{PRF}}(\lambda)=1].$$

Symmetric encryption. A symmetric key encryption scheme SE is a tuple of algorithms $(\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec})$. The setup algorithm produces the public parameters for the scheme, $pp \leftarrow\$ \mathsf{SE.Setup}(\lambda)$. The key generation algorithm outputs a shared symmetric key, $k \leftarrow\$ \mathsf{SE.Keygen}^{pp}()$. The encryption algorithm produces a ciphertext on an input message, $ct \leftarrow\$ \mathsf{SE.Enc}^{pp}(k,m)$, and the decryption algorithm decrypts the ciphertext to retrieve the enclosed message, $m \leftarrow\$ \mathsf{SE.Dec}^{pp}(k,ct)$. Correctness dictates that $\mathsf{SE.Dec}(dk, \mathsf{SE.Enc}(k,m)) = m$ for all valid keys k and messages m in the message space.

Indistinguishability for real-or-random ciphertexts under chosen-plaintext attacks (RoR-CPA) for a symmetric key encryption scheme SE is defined by the security game RORCPA in which an adversary is tasked with distinguishing a challenge ciphertext as a string of random bits or an encryption of a self-chosen plaintext; we consider a multi-key variant. The advantage of an adversary is defined as $\mathbf{Adv}^{\text{rorcpa}}_{\mathsf{SE},\mathcal{A},n}(\lambda) = \left|\Pr[\mathrm{RORCPA}^{\mathcal{A},1}_{\mathsf{SE},n}(\lambda)=1] - \mathrm{RORCPA}^{\mathcal{A},0}_{\mathsf{SE},n}(\lambda)=1]\right|$.

Public-key encryption. A public key encryption scheme PKE is a tuple of algorithms $(\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec})$. The setup algorithm produces the public parameters for the scheme, $pp \leftarrow\$ \mathsf{PKE.Setup}(\lambda)$. The key generation algorithm outputs a public encryption key and a se-

Game $\mathrm{INDCPA}^{\mathcal{A},b}_{\mathsf{PKE}}(\lambda)$
$pp \leftarrow\$ \mathsf{PKE.Setup}(\lambda)$
$(pk, sk) \leftarrow\$ \mathsf{PKE.Keygen}^{pp}()$
$b' \leftarrow\$ \mathcal{A}^{\mathrm{Enc}}(pk)$
Return b'
$\mathrm{Enc}(m_0, m_1)$
Return $\mathsf{PKE.Enc}(sk, m_b)$

$\mathsf{ElG.Setup}(\lambda)$	$\mathsf{ElG.Enc}^{pp}(ek, m)$
$(p, g, \mathbb{G}) \leftarrow\$ \mathsf{GGen}(\lambda)$	$r \leftarrow\$ \mathbb{Z}_p$
$pp \leftarrow (p, g, \mathbb{G})$	$ct \leftarrow (g^r, mek^r)$
Return pp	Return ct
$\mathsf{ElG.Keygen}^{pp}()$	$\mathsf{ElG.Dec}^{pp}(dk, ct)$
$x \leftarrow\$ \mathbb{Z}_p$	$(ct_1, ct_2) \leftarrow ct$
Return (g^x, x)	Return ct_2/ct_1^{dk}

Figure 2.3: Security games for public key encryption and useful construction.

cret decryption key, $(ek, dk) \leftarrow\$ \mathsf{PKE.Keygen}^{pp}()$. The encryption algorithm produces a ciphertext on an input message, $ct \leftarrow\$ \mathsf{PKE.Enc}^{pp}(ek, m)$, and the decryption algorithm decrypts the ciphertext to retrieve the enclosed message, $m \leftarrow\$ \mathsf{PKE.Dec}^{pp}(dk, ct)$. Correctness dictates that $\mathsf{PKE.Dec}(dk, \mathsf{PKE.Enc}(ek, m)) = m$ for all valid key pairs (ek, dk) and messages m in the message space.

Indistinguishability under chosen-plaintext attacks (IND-CPA) for a public key encryption scheme PKE is defined by the security game $\mathrm{INDCPA}^{\mathcal{A},b}_{\mathsf{PKE}}(\lambda)$ in which an adversary is tasked with distinguishing the decryption of a challenge ciphertext to one of two distinct self-chosen plaintexts. The advantage of an adversary is defined as $\mathbf{Adv}^{\mathrm{indcpa}}_{\mathsf{PKE},\mathcal{A}}(\lambda) = \left|\Pr[\mathrm{INDCPA}^{\mathcal{A},1}_{\mathsf{PKE}}(\lambda) = 1] - \mathrm{INDCPA}^{\mathcal{A},0}_{\mathsf{PKE}}(\lambda) = 1]\right|$.

We provide pseudocode for the ElGamal public key encryption scheme ElG below. It is IND-CPA-secure under the DDH assumption [TY98].

Proofs (and signatures) of knowledge. We define a non-interactive proof system Π over an efficiently computable relation R defined over pairs (x, w) where x is called the *statement* and w is called the *witness*. Let $\mathcal{L}$ be the language consisting of statements in R.

A non-interactive proof system Π is made up of the following algorithms. The setup algorithm produces the public parameters for execution, $pp \leftarrow\$ \Pi.\mathsf{Setup}(\lambda)$. The proving algorithm takes a witness and statement and produces a proof, $\pi \leftarrow\$ \Pi.\mathsf{Prove}^{pp}(w, x)$. The verification algorithm verifies a proof for a statement, $b \leftarrow \Pi.\mathsf{Ver}^{pp}(x, \pi)$. We further extend the notion of a non-interactive proof system to a signature of knowledge proof system Σ by modifying the proving and verification algorithms to support binding a message [GM17, CL06]. A signature of knowledge is similar to a

digital signature in that a message can only be validly signed with respect to a statement by a party with knowledge of a witness. The signing algorithm and signature verification algorithm additionally take a message m as input, $\Sigma.\mathsf{Prove}^{pp}(w, x, m)$ and $\Sigma.\mathsf{Ver}^{pp}(x, m, \pi)$.

The below definitions will apply to both a non-interactive proof system and to a signature of knowledge proof system. Extensions to the non-interactive proof system definitions introduced for signatures of knowledge are highlighted.

Completeness. A proof system is *complete* if given a true statement, a prover with a witness can convince the verifier. We will make use of a proof system with perfect completeness. A proof system has *perfect completeness* if for all $(x, w) \in \mathsf{R}$ and all m in the message space,

$$\Pr[\Pi/\Sigma.\mathsf{Ver}(x, m, \Pi/\Sigma.\mathsf{Prove}(w, x, m)) = 1] = 1 .$$

Knowledge soundness. A proof system is computationally *knowledge sound* if whenever a prover is able to produce a valid proof, it is possible to extract a valid witness from the prover's internal transcript. The prover's internal transcript, denoted by τ, contains the description of the prover algorithm and input along with any random choices made. Knowledge soundness is defined by the security game SOUND in which an adversary is tasked with finding a verifying statement and proof for which the extractor does not extract a valid witness. The advantage of an adversary is defined as $\mathbf{Adv}^{\text{sound}}_{\Pi,\mathsf{X},\mathcal{A}}(\lambda) = \Pr[\text{SOUND}^{\mathcal{A}}_{\Pi,\mathsf{X}}(\lambda) = 1]$.

Zero knowledge. A proof system is computationally *zero-knowledge* if a proof does not leak any information besides the truth of a statement. Zero knowledge is defined by the security game ZK in which an adversary is tasked with distinguishing between proofs generated from a valid witness and simulated proofs generated without a witness. The advantage of an adversary is defined as $\mathbf{Adv}^{\text{zk}}_{\Pi,\mathsf{S},\mathcal{A}}(\lambda) = \left|\Pr[\text{ZK}^{\mathcal{A},1}_{\Pi,\mathsf{S}}(\lambda) = 1] - \text{ZK}^{\mathcal{A},0}_{\Pi,\mathsf{S}}(\lambda) = 1]\right|$, with respect to simulator algorithm S.

Simulation extractability. Simulating a proof for a false statement might jeopardize the soundness of the proof system. It may be possible for an adversary to modify the proof into another proof for a false instance. This scenario is common in security proofs of cryptographic schemes, in which case it is desireable to have some sort of non-malleable property that prevents this type of break in soundness even in the presence of simulated proofs.

A proof system is *simulation extractable* if even after seeing many simulated proofs, whenever

Game $\text{SOUND}^{\mathcal{A}}_{\Pi,\mathsf{X}}(\lambda)$; $\text{SOUND}^{\mathcal{A}}_{\Sigma,\mathsf{X}}(\lambda)$
$pp \leftarrow\!\!\$\; \Pi/\Sigma.\mathsf{Setup}(\lambda)$
$(x, \pi, m) \leftarrow\!\!\$\; \mathcal{A}(pp)$
$w \leftarrow \mathsf{X}(\tau_{\mathcal{A}})$
$b \leftarrow \Pi/\Sigma.\mathsf{Ver}(x, m, \pi)$
Return $\bigwedge \begin{pmatrix} \Pi/\Sigma.\mathsf{Ver}(x, m, \pi) \\ (x, w) \notin \mathsf{R} \end{pmatrix}$

Game $\text{ZK}^{\mathcal{A},b}_{\Pi,\mathsf{S}}(\lambda)$; $\text{ZK}^{\mathcal{A},b}_{\Sigma,\mathsf{S}}(\lambda)$
$pp_1 \leftarrow\!\!\$\; \Pi/\Sigma.\mathsf{Setup}(\lambda)$
$(pp_0, \xi) \leftarrow\!\!\$\; \mathsf{S}.\mathsf{Setup}(\lambda)$
$b' \leftarrow\!\!\$\; \mathcal{A}^{\text{PROVE}}(pp_b)$
Return b'
$\text{PROVE}(x, w, m)$
Require $(x, w) \in \mathcal{R}$
$\pi_1 \leftarrow\!\!\$\; \Pi/\Sigma.\mathsf{Prove}(x, w, m)$
$\pi_0 \leftarrow\!\!\$\; \mathsf{S}.\mathsf{Prove}(\xi, x, m)$
Return π_b

Game $\text{SIMEXT}^{\mathcal{A}}_{\Pi,\mathsf{X},\mathsf{S}}(\lambda)$; $\text{SIMEXT}^{\mathcal{A}}_{\Sigma,\mathsf{X},\mathsf{S}}(\lambda)$
$pp_1 \leftarrow\!\!\$\; \Pi/\Sigma.\mathsf{Setup}(\lambda)$
$(pp_0, \xi) \leftarrow\!\!\$\; \mathsf{S}.\mathsf{Setup}(\lambda)$
$(x, \pi, m) \leftarrow\!\!\$\; \mathcal{A}^{\text{SIMPROVE}}(pp_b)$
$w \leftarrow \mathsf{X}(\tau_{\mathcal{A}})$
Return $\bigwedge \begin{pmatrix} \Pi/\Sigma.\mathsf{Ver}(x, m, \pi) \\ (x, w) \notin \mathsf{R} \\ (x, \pi, m) \notin \mathcal{Q} \end{pmatrix}$
$\text{SIMPROVE}(x, m)$
$\pi \leftarrow\!\!\$\; \mathsf{S}.\mathsf{Prove}(\xi, x, m)$
$\mathcal{Q} \leftarrow\!\!\cup\; (x, \pi, m)$
Return π

Figure 2.4: Security games for non-interactive zero-knowledge proofs of knowledge.

a prover produces a new proof, it is possible to extract a valid witness from their internal transcript. Simulation extractability is defined by the security game SIMEXT in which an adversary is given access to a simulation oracle and tasked with finding a verifying statement and proof for which the extractor does not extract a valid witness. The advantage of an adversary is defined as $\mathbf{Adv}^{\text{simext}}_{\Pi,\mathsf{X},\mathsf{S},\mathcal{A}}(\lambda) = \Pr[\text{SIMEXT}^{\mathcal{A}}_{\Pi,\mathsf{X},\mathsf{S}}(\lambda) = 1]$. Observe that simulation extractability implies knowledge soundness, since the games are identical if the simulation extractability adversary does not use its simulation oracle.

Authenticated dictionaries. An *authenticated dictionary* (AD) maintains and commits to a collection of key/value pairs $[(k_i, v_i)]_i$, where every key is unique, with a digest d. An initial digest and state are produced via $(d_0, st) \leftarrow \mathsf{Init}^{pp}()$ following a setup producing public parameters $pp \leftarrow\!\!\$\; \mathsf{Setup}(\lambda)$ where λ is a security parameter. The public parameters are included implicitly in all algorithms, and we may drop the superscript if the use is clear from context. A set of key-value mappings may be updated to produce a new digest, $(d', st) \leftarrow \mathsf{Upd}([(k_j, v_j)]_j : st)$. It provides proofs for key lookups, $(v, \pi) \leftarrow \mathsf{Lkup}(k : st)$, that can be verified given the digest commitment, $0/1 \leftarrow \mathsf{VerLkup}(d, k, v, \pi)$. An authenticated dictionary must satisfy *key binding*, which means that it is infeasible to produce valid lookup proofs for key k to different values v and v'. ADs can also be augmented with invariant update proofs, proving that a certain invariant $\Phi(k, v, v')$ is preserved for all keys during an update; we augment the Upd algorithm to additionally return a proof and provide an accompanying verification

Game $\textsc{Bind}_{\mathsf{AD}}^{\mathcal{A}}(\lambda)$
$pp \leftarrow\$ \mathsf{AD.Setup}(\lambda)$
$(k, d, (v_A, \pi_A), (v_B, \pi_B)) \leftarrow\$ \mathcal{A}(pp)$
Return $\bigwedge \begin{pmatrix} \mathsf{AD.VerLkup}(d, k, v_A, \pi_A) \\ \mathsf{AD.VerLkup}(d, k, v_B, \pi_B) \\ v_A \neq v_B \end{pmatrix}$

Game $\textsc{InvSound}_{\mathsf{AD},\Phi}^{\mathcal{A}}(\lambda)$
$pp \leftarrow\$ \mathsf{AD.Setup}(\lambda)$
$(k, (v_A, \pi_A), (v_B, \pi_B), [d_j]_j^m, [\pi_{\Phi,j}]_j^{m-1}) \leftarrow\$ \mathcal{A}(pp)$
Return $\bigwedge \begin{pmatrix} \mathsf{AD.VerLkup}(d_1, k, v_A, \pi_A) \\ \mathsf{AD.VerLkup}(d_m, k, v_B, \pi_B) \\ \left[\mathsf{AD.VerUpd}(d_j, d_{j+1}, \pi_{\Phi,j})\right]_j^{m-1} \\ \Phi(k, v_A, v_B) \neq 1 \end{pmatrix}$

Game $\textsc{Bind}_{\mathsf{VC}}^{\mathcal{A}}(\lambda)$
$pp \leftarrow\$ \mathsf{VC.Setup}(\lambda)$
$(d, st) \leftarrow\$ \mathcal{A}_1(pp)$
$\mathcal{D} \leftarrow \mathcal{D} \cup [d]$
$(i, (d_A, v_A, \pi_A), (d_B, v_B, \pi_B)) \leftarrow\$ \mathcal{A}_2^{\textsc{Prefix}}(pp)$
Return $\bigwedge \begin{pmatrix} \mathsf{VC.VerLkup}(d_A, i, v_A, \pi_A) \\ \mathsf{VC.VerLkup}(d_B, i, v_B, \pi_B) \\ v_A \neq v_B \\ d_A \in \mathcal{D} \wedge d_B \in \mathcal{D} \end{pmatrix}$

Oracle $\textsc{Prefix}(d, d', j, \pi)$
Require $d \in \mathcal{D}$
If $\mathsf{VC.VerUpd}(d, d', j, \pi)$ then
$\quad \mathcal{D} \leftarrow \mathcal{D} \cup [d']$

Figure 2.5: Security games for strong key binding (left) and invariant preservation of updates (middle) for authenticated dictionaries. Security game for index binding (right) for append-only vector commitments.

algorithm, $0/1 \leftarrow \mathsf{VerUpd}(d, d', \pi)$. The invariant proof must satisfy soundness, meaning if the verification algorithm succeeds, the invariant is preserved. We will primarily be concerned with the versioned invariant which has been previously used in Merkle trees [MBB$^+$15, Bon16].

The most prevalent authenticated dictionary implementations in practice are based on *Merkle trees* [Mer87, MBB$^+$15, PP15]. Merkle trees admit lookup proofs and update proofs for a single key which are of size and verification time $\mathcal{O}(\log N)$ for dictionaries of size N.

Figure 2.5 provides the security games for the strong key binding and invariant update soundness properties of authenticated dictionaries. The binding game requires an adversary to output two lookup proofs for different values that verify under the same key for the same digest. The invariant update soundness game requires an adversary to provide valid lookups for a key that does not satisfy the invariant across two digests, while also providing a sequence of invariant proofs that the invariant is preserved across the two digests. The invariant is defined as a boolean function Φ that takes as input a key, initial value, and updated value, then outputs 1 if the invariant is satisfied. We define an adversary's advantage against these games, respectively, as:

$$\mathbf{Adv}_{\mathsf{AD},\mathcal{A}}^{\mathsf{bind}}(\lambda) = \Pr[\textsc{Bind}_{\mathcal{A}}^{\mathsf{AD}}(\lambda) = 1] \,, \qquad \mathbf{Adv}_{\mathsf{AD},\Phi,\mathcal{A}}^{\mathsf{inv}}(\lambda) = \Pr[\textsc{InvSound}_{\mathcal{A}}^{\mathsf{AD},\Phi}(\lambda) = 1] \,.$$

In this work, we focus on the versioned invariant. The versioned invariant Φ_{vsn} parses values as a value-version tuple (v, u). It enforces (1) the key's version number does not decrease, and (2) two

different values for a key cannot be shown for the same version number. It is defined as follows:

$$\Phi_{\mathsf{vsn}}(k,(v_A,u_A),(v_B,u_B)) = u_A < u_B \lor (u_A = u_B \land v_A = v_B) \; .$$

Some applications require a stronger invariant to be maintained among mapped values in an AD, which we will refer to as the *append-only* invariant. In the append-only invariant, values of an AD are parsed as lists of values $L = [v_j]_j^{\ell}$. The invariant enforces that the list can only be appended to, i.e., previous values in the list do not change. More precisely, we define Φ_{app} as follows:

$$\Phi_{\mathsf{app}}(k, L_A = [v_{A,j}]_j^{\ell_A}, L_B = [v_{B,j}]_j^{\ell_B}) = \ell_A \le \ell_B \land \bigwedge_j^{\ell_A} v_{A,j} = v_{B,j} \; .$$

Append-only vector commitments. A *vector commitment* (VC) commits to an ordered list of elements $[v_i]_i$. Setup and initialization syntax follow the same as for ADs. An *append-only* VC provides an update algorithm to append elements to the end of the list, $(d', st) \leftarrow \mathsf{Upd}([v'_i]_i : st)$, as well as supports efficient prefix proofs that a commitment commits to a prefix of another: $\pi \leftarrow \mathsf{ProveUpd}(j : st)$ and $0/1 \leftarrow \mathsf{VerUpd}(d', d, j, \pi)$ where $L[0:j] = L'$ for list L and L' corresponding to digests d and d', respectively. A VC supports efficient lookups with proof of elements by index, $(v, \pi) \leftarrow \mathsf{Lkup}(i : st)$ with accompanying verification algorithm $0/1 \leftarrow \mathsf{VerLkup}(d, i, v, \pi)$. A VC must satisfy *index binding* meaning that it should not be possible to provide valid lookup proofs to different values for the same index. Again, append-only VCs can be derived from Merkle trees [CW09, MKL+20, BKLZ20]; it supports lookup proofs and arbitrary-length update proofs of size and verification time $\mathcal{O}(\log N)$ for vectors of size N.

Figure 2.5 (right) provides the pseudocode security game defining *index binding* for append-only vector commitments. The game requires an adversary to produce two valid lookup proofs to the same index for different values. The adversary is allowed to give lookup proofs for different digests as long as they additionally prove that the two digests share prefixes. We define an adversary's advantage against index binding as:

$$\mathbf{Adv}^{\mathsf{bind}}_{\mathsf{VC},\mathcal{A}}(\lambda) = \Pr[\textsc{Bind}^{\mathsf{VC}}_{\mathcal{A}}(\lambda) = 1] \; .$$

CHAPTER 3

SECURE REPORTING FOR CONTENT MODERATION

Billions of users communicate via private messaging on platforms like Facebook, Twitter, and Signal. Their success means these platforms are increasingly used for large-scale spam, harassment, and propagation of misinformation. One way platform operators address these threats is via *content moderation*: the receiver of a message can report it to a moderator. If the moderator determines (via human judgment, machine learning algorithm, or both) that the message violated the platform's policies, the platform can ban its sender.

To ensure moderation is not itself abused, the platform must be able to verify both the content of the reported message and associated metadata, e.g. the sender and receiver identity. Doing this is challenging for end-to-end (E2E) encrypted messaging because the platform does not see the plaintext messages.

As cryptography was used to hide message content, it can also be used to provide message authenticity for reports. A naive solution for moderation in the encrypted setting is to require per-message digital signatures. These provide accountability: even if the moderator does not initially see the message content, a digital signature of the message will convince the moderator of a user's authorship.

However, digital signatures violate the privacy that users expect from messaging systems: a moderator or their communicating partner may post their messages publicly. Just as a digital signature convinces the moderator of a user's authorship, it will also convince any other party to which the signature is shared. Our modern private messaging systems are carefully designed to protect against such attacks on user privacy by providing *deniability* [BGB04, PM16].

The first forays into addressing moderation with appropriate user deniability stem from Facebook's pioneering efforts on *message franking* [Fac17, GLR17]. Message franking has two main components. First, the E2E encryption uses specially-constructed ciphertexts that include a compact commitment to the plaintext. Second, the platform cryptographically binds the sender and receiver identities to the ciphertexts using a reporting tag (concretely, a MAC over the relevant metadata and the commitment). This "message franks" or the MAC over the metadata and commitment are deniable in that they could have been forged using the moderator's keys. Because this approach only

uses symmetric-key cryptography, we call it *symmetric* message franking (SMF).

SMF carefully navigates the three security requirements of content moderation for E2E encrypted messaging. First, messages not included in reports should remain private. Second, moderation should achieve *accountability*: given a reported message and sender identity, the moderator should always be able to verify the sender sent that message. Without accountability, moderation can itself be abused: malicious receivers can target senders with fake reports, or malicious senders can send messages that cannot be reported. Finally, moderation for E2E-encrypted messages should be *deniable*: only the moderator should be able to verify the report. User messages are privy only to their communication partner and the moderator; this protects users from backlash or embarrassment if their messages are posted publicly after a compromise.

While SMF offers an efficient solution to the problem of abuse reporting, this chapter will explore two messaging and abuse settings in which SMF falls short. The first is metadata-private messaging. Some modern messaging protocols, such as "sealed sender" [Lun17] introduced by Signal, hides the communication metadata of who is communicating with whom from the platform. SMF relies on this metadata to be available to the platform so that it may be included in the message frank. The second is abuse of misinformation campaigns and viral forwarding. SMF reveals the sender of a reported message, but does not help with identifying the source of a forwarded message, or the set of users to which it was ultimately sent. In each of these settings, new cryptographic techniques are needed to address the privacy and accountability goals that are raised.

3.1 Asymmetric Message Franking

As described above, SMF cannot be used for moderation when it is impossible to associate identities to encrypted messages. One such setting is *metadata-private* messaging, depicted in the middle diagram of Figure 3.1. Metadata-private messaging systems not only use E2E encryption, but also hide the sender and/or receiver identities of messages from the platform. In these systems the platform knows the identities of all registered parties but does not learn those identities during communication, for example, as in Signal's sealed sender feature. Similarly, one may consider decentralized or federated settings where the moderator is decoupled from the platform. (See the right-hand diagram of Figure 3.1.) Third-party moderation is necessary in decentralized or federated messaging systems like Matrix or Mastodon. In such systems no single party operates the platform, so the moderator

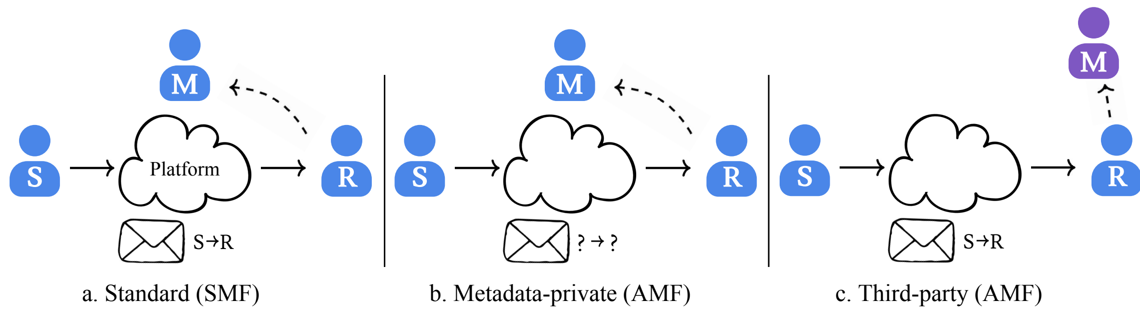

Figure 3.1: Settings for content moderation of messaging. The solid arrow denotes sending a message across the platform and the dashed arrow denotes reporting a message to the moderator. In the standard setting, messages sent across the platform are associated with sender and receiver identities and the platform is the moderator. In the metadata-private setting, the associated sender and receiver identities of messages are hidden from the platform, and by extension, the moderator. In the third-party setting, the moderator is separate from the platform, and thus also cannot associate sender and receiver identities to messages. Our AMF primitive targets the latter two settings.

must be distinct. Even in centralized systems like Twitter, third-party moderation is advantageous if the platform cannot adequately moderate messages, or if sub-communities want to enforce their own content policies. Allowing the moderator to be distinct can also enable cross-platform moderation of multiple messaging systems. As in the metadata-private setting, a third-party moderator does not learn the needed sender and receiver identities associated with messages. In metadata-private settings, this information is cryptographically hidden, whereas in third-party settings it is simply unavailable because the moderator doesn't run the identity infrastructure. Because AMF schemes are public-key, they can be used in conjunction with PKI to build third-party moderation.

A naive solution for moderation in these settings is per-message digital signatures. These provide accountability even if the moderator cannot see metadata or messages, but not deniability: anyone can verify signatures, not just the moderator. Indeed, this and other approaches based on existing primitives fail because of a fundamental tension between accountability and deniability. To make moderation a reality for metadata-private messaging and other settings, new cryptography is needed.

This work defines and constructs asymmetric message franking (AMF) schemes. AMF schemes are special signatures in which a sender signs a message so that only one of two designated parties, the receiver or the moderator, can verify it. The signature also proves to the receiver that the signature can be verified by the moderator. AMF schemes are deniable and do not require the platform to associate identities with encrypted messages. Thus, AMF schemes resolve the main technical barrier

to content moderation for metadata-private messaging.

3.1.1 Meaningful Deniability in Messaging

We want AMFs to provide deniability in the event that keys or messages are posted publicly after a compromise. Our setting is therefore most similar to the deniability guarantees sought for designated verifier signatures and proofs [JSI96], but different than settings that allow one to deny encrypted message contents even to an eavesdropper that sees all traffic [CDNO97]. An adversary that observes the actual transmission of a message or ciphertext is totally convinced of its origin in our setting. Instead, our concern is not this adversary's conviction, but its ability to convince others. As long as the attacker cannot use what it learns through network manipulation or endpoint compromise to convince others, we have achieved deniability.

The types of deniability guarantees we target have long been a goal in various contexts [CA89, Cha90], including messaging [Fac17]. The inability to prevent major compromises has made lack of deniability an increasingly pressing concern. In the 2016 United States' and 2017 French presidential elections, certain candidates' systems were compromised and sensitive data was dumped publicly online. DKIM email signatures prevented the Clinton campaign from denying authorship for hacked emails posted by Wikileaks in 2016. In contrast, in 2017 the Macron campaign was able to effectively deny the authenticity of leaked messages by including decoy messages as a countermeasure. This defense was only possible because of a lack of cryptographic evidence. One result of these breaches is that politicians and others increasingly use E2E encrypted messaging systems that provide deniability. If E2E encryption provides deniability, the cryptography used for moderation must preserve this deniability. This is a crucial reason why AMFs must be deniable.

These examples additionally demonstrate that deniability in messaging is practically important: it is necessary, but not always sufficient, for (what we call) social deniability, i.e., that people are convinced by a denial [RMA+23]. Our goal is to ensure that whatever prior belief people have about the likelihood a message is valid should remain unchanged by the use of cryptography, and to have a system that works with other techniques for increasing the success of social deniability (e.g., use of decoys). We do note that because of pervasive propaganda campaigns an awareness has developed among the general public that malicious parties will try to influence popular sentiment by forging

content. This would seem to make social deniability more feasible, as people are unlikely to be convinced by an unverified attribution in the era of "fake news".

An important implication of all this is that, to issue a denial that will convince the general public, it is not sufficient to demonstrate the (perhaps non-constructive) *existence* of a forger who *could* have forged a message—there must exist concrete and runnable forgery algorithms that could have been used by influence campaigns or other adversaries. Our eventual construction has three such implemented algorithms for different compromise scenarios.

3.1.2 Syntax and Security Notions

We introduce a new primitive, *asymmetric message franking* (AMF), that provides the cryptographic algorithms needed for secure metadata-private moderation. We will present the algorithms and security definitions of an AMF scheme in three parts. First, we present a brief preliminary on key generation. Then, we describe the accountability algorithms and definitions. Finally, we present the three algorithms used for deniability and definitions. We choose to decouple our security treatment of AMFs from the accompanying end-to-end (E2E) encryption scheme to simplify and modularize the analysis.

Formally, an asymmetric message franking scheme $\mathsf{AMF} = (\mathsf{Keygen}, \mathsf{Frank}, \mathsf{Verify}, \mathsf{Judge}, \mathsf{Forge}, \mathsf{RForge}, \mathsf{JForge})$ is a tuple of seven algorithms. An AMF scheme is associated with a public key space PK, secret key space SK, message space M, and signature space Σ. To simplify notation of inputs in the algorithms, we assume all pk inputs are in PK, all sk inputs are in SK, all m inputs are in M, and all σ inputs are in Σ.

AMF key generation. Following a setup to generate global public parameters $pp \leftarrow\$ \mathsf{Setup}(\lambda)$, AMF key generation, $(pk, sk) \leftarrow\$ \mathsf{Keygen}(pp)$, is a randomized key generation algorithm which outputs a public key pair $(pk, sk) \in \mathsf{PK} \times \mathsf{SK}$. We assume the public key pk can be uniquely recovered from the private key sk. Our schemes have this property. We also assume for simplicity that the judge, senders, and receivers all use the same key generation algorithm.

We will assume that key pairs can be confirmed to be valid. More precisely, we will require a deterministic algorithm valid $\mathsf{PK} \times \mathsf{SK} \rightarrow \{0, 1\}$ which takes as input a key pair $(pk, sk) \in \mathsf{PK} \times \mathsf{SK}$ and outputs a bit b denoting whether the key pair is a valid pair ($b = 1$) or not ($b = 0$). The purpose of

this procedure is to verify that a (possibly adversarially chosen) key pair is well-formed relative to some relationship between pk and sk. In our schemes this will be a single exponentiation.

Accountability Notions

For an AMF = (Setup, Keygen, Frank, Verify, Judge, Forge, RForge, JForge), the three accountability algorithms are Frank, Verify, and Judge. These algorithms are used for creating and verifying signatures. We explain the syntax of each algorithm in turn, then describe the corresponding accountability security notions.

- $\sigma \leftarrow\!\$\, \mathsf{Frank}(sk_s, pk_r, pk_j, m)$: The (randomized) message signing or *franking* algorithm takes as input a receiver public key pk_r, a judge public key pk_j, a sender secret key sk_s, and a message m. It outputs a signature σ.
- $b \leftarrow \mathsf{Verify}(pk_s, sk_r, pk_j, m, \sigma)$: The deterministic receiver verification algorithm takes as input a sender public key pk_s, receiver secret key sk_r, judge public key pk_j, message m, and signature σ, then outputs a bit. The receiver runs this to ensure the message, signature pair (m, σ) is well-formed and reportable to the judge.
- $b \leftarrow \mathsf{Judge}(pk_s, pk_r, sk_j, m, \sigma)$: The deterministic judge authentication algorithm takes as input a sender public key pk_s, receiver public key pk_r, judge secret key sk_j, message m, and signature σ, then outputs a bit. This algorithm is used by the judge to check the authenticity of reported messages, ensuring the message was really sent from the sender and was meant for the recipient.

This formalization restricts attention to non-interactive schemes for which franking, verification, and judging requires sending just a single message. Such non-interactive schemes have important practical benefits, but it is conceivable that there might be some benefits of generalizing our treatment to include interactive schemes, which we leave for future work.

Correctness. Informally, we require AMF signatures created by the franking algorithm are both verified and judged successfully. Formally, for all messages ,m, and for all pairs of public keys, $(pk_{\{s,r,j\}}, sk_{\{s,r,j\}})$, it holds that

$$\Pr\left[\mathsf{Verify}(pk_s, sk_r, pk_j, m, \mathsf{Frank}(sk_s, pk_r, pk_j, m)) = 1\right] = 1$$

Game $\mathrm{RBind}^{\mathcal{A}}_{\mathsf{AMF}}(\lambda)$	Game $\mathrm{SBind}^{\mathcal{A}}_{\mathsf{AMF}}(\lambda)$
$pp \leftarrow\$ \mathsf{AMF.Setup}(\lambda)$	$pp \leftarrow\$ \mathsf{AMF.Setup}(\lambda)$
$(pk_s, sk_s) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$	$(pk_r, sk_r) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_j, sk_j) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$	$(pk_j, sk_j) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_r, m, \sigma) \leftarrow \mathcal{A}^{\mathrm{Frank,Judge}}(pk_s, pk_j)$	$(pk_s, m, \sigma) \leftarrow \mathcal{A}^{\mathrm{Verify,Judge}}(pk_r, pk_j)$
Return $\bigwedge \begin{pmatrix} (pk_r, pk_j, m) \notin \mathcal{Q} \\ \mathsf{AMF.Judge}(pk_s, pk_r, sk_j, m, \sigma) \end{pmatrix}$	Return $\bigwedge \begin{pmatrix} \mathsf{AMF.Verify}(pk_s, sk_r, pk_j, m, \sigma) \\ \neg\mathsf{AMF.Judge}(pk_s, pk_r, sk_j, m, \sigma) \end{pmatrix}$
Oracle $\mathrm{Frank}(pk'_r, pk'_j, m)$	Oracle $\mathrm{Verify}(pk'_s, pk'_j, m, \sigma)$
$\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(pk'_r, pk'_j, m)\}$	Return $\mathsf{AMF.Verify}(pk'_s, sk_r, pk'_j, m, \sigma)$
Return $\mathsf{AMF.Frank}(sk_s, pk'_r, pk'_j, m)$	
Oracle $\mathrm{Judge}(pk'_s, pk'_r, m, \sigma)$	Oracle $\mathrm{Judge}(pk'_s, pk'_r, m, \sigma)$
Return $\mathsf{AMF.Judge}(pk'_s, pk'_r, sk_j, m, \sigma)$	Return $\mathsf{AMF.Judge}(pk'_s, pk'_r, sk_j, m, \sigma)$

Figure 3.2: Accountability games for AMF schemes: receiver binding (left) and sender binding (right).

and

$$\Pr\left[\mathsf{Judge}(pk_s, pk_r, sk_j, m, \mathsf{Frank}(sk_s, pk_r, pk_j, m)) = 1\right] = 1$$

where the probabilities are taken over the random coins used in Frank.

Security notions for accountability. First and foremost an AMF scheme should prevent a party from impersonating a sender to a receiver. This goal, which we call *unforgeability*, is a lifting of standard digital signature unforgeability to the setting of AMF schemes. AMFs should also (1) prevent any sender from creating a signature that can be verified by the receiver but not the moderator, and (2) prevent any receiver from framing a sender by creating a signature on a message that wasn't sent. Following the terminology used in symmetric message franking [GLR17] we refer to these goals as *sender binding* and *receiver binding*, respectively.

It turns out sender binding and receiver binding together imply unforgeability. In this section, we proceed by formalizing the sender binding and receiver binding accountability notions.

Receiver binding is specified formally in game RBind on the left-hand side of Figure 3.2. The adversary plays the role of a reciever and attempts to create a signature that from a sender pk_s to an adversarially chosen pk_r that correctly judges by pk_j. The adversary is given a Frank oracle for some (honest) sender, to which it can query messages signed to chosen receiver and judge public

keys. We also give the adversary access to a Judge oracle to query chosen message and signature pairs. It tries to output a message and signature, distinct from all Frank oracle outputs, for which Judge outputs 1. For an adversary $\mathcal{A}$ and message franking scheme AMF we define the RBIND advantage of $\mathcal{A}$ against AMF as $\mathbf{Adv}^{\text{rbind}}_{\mathsf{AMF},\mathcal{A}}(\lambda) = \Pr\left[\text{RBIND}^{\mathcal{A}}_{\mathsf{AMF}}(\lambda) = 1\right]$ where the probability here (and for subsequent use of games) is over all the random coins used in the game, including those of the adversary.

Sender binding is specified formally in game SBIND on the right-hand side of Figure 3.2. The adversary plays the role of a sender and its goal is to generate, for some adversarially chosen pk_s, an AMF signature that Verify validates but Judge rejects with pk_r and pk_j. The adversary is given a pair of oracles for Verify and Judge to which it can query message and signature pairs. For an adversary $\mathcal{A}$ and message franking scheme AMF we define the SBIND advantage of $\mathcal{A}$ against AMF as $\mathbf{Adv}^{\text{sbind}}_{\mathsf{AMF},\mathcal{A}}(\lambda) = \Pr\left[\text{SBIND}^{\mathcal{A}}_{\mathsf{AMF}} = 1\right]$.

Deniability Notions

To support deniability, we equip AMF schemes with three deniability algorithms and associate to each a security notion. We include the forging algorithms as part of the scheme to emphasize their importance in providing practically-meaningful deniability guarantees. They will be efficient to execute and as easy to implement as the other algorithms. The deniability algorithms for an AMF scheme AMF are Forge, RForge, and JForge. We give a formal description of each along with some intuition about the deniability setting they correspond to.

Universal deniability requires that any non-participating party (no access to sender, receiver, or judge secret keys) can forge a signature that is indistinguishable from honestly-generated signatures to other non-participating parties. Intuitively, this allows the sender to claim a message originated from any non-participating party. This is the purpose of the Forge algorithm of an AMF scheme.

- $\sigma \leftarrow\$ \mathsf{Forge}(pk_s, pk_r, pk_j, m)$: The forge algorithm takes a sender public key pk_s, receiver public key pk_r, a judge public key pk_j, and a message m, then outputs a “forged” AMF signature σ.

We formalize universal deniability in game UDEN, the leftmost in Figure 3.3. The adversary is given access to a frank oracle that outputs a signature created from Frank or Forge depending on a challenge bit that is the adversary’s goal to guess. In this deniability game and all subsequent deniability games,

Game $\mathrm{UDen}^{\mathcal{A},b}_{\mathsf{AMF}}(\lambda)$

$pp \leftarrow\$ \mathsf{AMF.Setup}(\lambda)$
$(pk_s, sk_s) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_r, sk_r) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_j, sk_j) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$b' \leftarrow \mathcal{A}^{\mathrm{Frank}}(sk_s, pk_r, pk_j)$
Return b'

Oracle $\mathrm{Frank}(m)$

$\sigma_0 \leftarrow \mathsf{AMF.Frank}(sk_s, pk_r, pk_j, m)$
$\sigma_1 \leftarrow \mathsf{AMF.Forge}(pk_s, pk_r, pk_j, m)$
Return σ_b

Game $\mathrm{RCompDen}^{\mathcal{A},b}_{\mathsf{AMF}}(\lambda)$

$pp \leftarrow\$ \mathsf{AMF.Setup}(\lambda)$
$(pk_s, sk_s) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_j, sk_j) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_r, sk_r, aux) \leftarrow \mathcal{A}_1(pk_s, pk_j)$
Return $\bigwedge \begin{pmatrix} \mathsf{valid}(pk_r, sk_r) \\ \mathcal{A}_2^{\mathrm{Frank}}(sk_s, sk_r, pk_j, aux) \end{pmatrix}$

Oracle $\mathrm{Frank}(m)$

$\sigma_0 \leftarrow \mathsf{AMF.Frank}(sk_s, pk_r, pk_j, m)$
$\sigma_1 \leftarrow \mathsf{AMF.RForge}(pk_s, sk_r, pk_j, m)$
Return σ_b

Game $\mathrm{JCompDen}^{\mathcal{A},b}_{\mathsf{AMF}}(\lambda)$

$pp \leftarrow\$ \mathsf{AMF.Setup}(\lambda)$
$(pk_s, sk_s) \leftarrow\$ \mathsf{AMF.Keygen}(pp)$
$(pk_r, sk_r, pk_j, sk_j, aux) \leftarrow \mathcal{A}_1(pk_s)$
Return $\bigwedge \begin{pmatrix} \mathsf{valid}(pk_r, sk_r) \\ \mathsf{valid}(pk_j, sk_j) \\ \mathcal{A}_2^{\mathrm{Frank}}(sk_s, sk_r, sk_j, aux) \end{pmatrix}$

Oracle $\mathrm{Frank}(m)$

$\sigma_0 \leftarrow \mathsf{AMF.Frank}(sk_s, pk_r, pk_j, m)$
$\sigma_1 \leftarrow \mathsf{AMF.JForge}(pk_s, pk_r, sk_j, m)$
Return σ_b

Figure 3.3: Deniability security games for AMF schemes: universal deniability (left), receiver compromise deniability (middle), and judge compromise deniability (right).

the adversary is given access to the sender's secret key sk_s to model sender compromise. For an adversary $\mathcal{A}$ and asymmetric message franking scheme AMF we define the UDen advantage of $\mathcal{A}$ against AMF as

$$\mathbf{Adv}^{\mathsf{uden}}_{\mathsf{AMF},\mathcal{A}}(\lambda) = \left| \Pr\left[\mathrm{UDen}^{\mathcal{A},1}_{\mathsf{AMF}}(\lambda) = 1\right] - \Pr\left[\mathrm{UDen}^{\mathcal{A},0}_{\mathsf{AMF}}(\lambda) = 1\right] \right| .$$

Receiver compromise deniability requires that a party with access to the receiver's secret key can forge a signature that is indistinguishable from honestly-generated signatures to other parties with access to the receiver's secret key. This captures deniability in the case where the receiver's secret key is compromised, and allows the sender to claim a message originates from a compromising party or malicious receiver. The RForge algorithm is used for receiver compromise deniability.

- $\sigma \leftarrow\$ \mathsf{RForge}(pk_s, sk_r, pk_j, m)$: The receiver forge algorithm takes a sender public key pk_s, receiver secret key sk_r, a judge public key pk_j, and a message m, then outputs a "forged" AMF signature σ.

We formalize receiver compromise deniability in two-stage game RCompDen, the middle game in Figure 3.3. The second-stage adversary $\mathcal{A}_2$ is given access to a frank oracle that outputs a signature created from Frank or RForge depending on a challenge bit. The goal is to guess the challenge bit given the sender and receiver secret keys, sk_s and sk_r. We strengthen the definition by answering the frank oracle queries using a public key pair for the receiver generated in the first stage by adversary

$\mathcal{A}_1$. For an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and message franking scheme AMF, we define the RCOMPDEN advantage of $\mathcal{A}$ against AMF as

$$\mathbf{Adv}^{\text{rden}}_{\mathsf{AMF},\mathcal{A}}(\lambda) = \left|\Pr\left[\text{RCOMPDEN}^{\mathcal{A},1}_{\mathsf{AMF}}(\lambda) = 1\right] - \Pr\left[\text{RCOMPDEN}^{\mathcal{A},0}_{\mathsf{AMF}}(\lambda) = 1\right]\right| .$$

Judge compromise deniability requires that a party with access to the judge's secret key can forge a signature that is indistinguishable from honestly-generated signatures to other parties even with access to the judge's secret key and receiver's secret key. This captures deniability in the case where the judge's secret key has become compromised, and allows the sender to claim a message originates from a compromising party or malicious judge. Our definition maintains deniability even in the case where the receiver's secret key is compromised as well. We discuss alternate, weaker deniability notions at the end of this section. The JForge algorithm is used for judge compromise deniability.

- $\sigma \leftarrow\!\!\$\ \mathsf{JForge}(pk_s, pk_r, sk_j, m)$: The judge forge algorithm takes a sender public key pk_s, receiver public key pk_r, a judge secret key sk_j, and a message m, then outputs a "forged" AMF signature σ.

We formalize judge compromise deniability in two-stage game JCOMPDEN, the right-most game in Figure 3.3. The second-stage adversary $\mathcal{A}_2$ is given access to a frank oracle that outputs a signature created from Frank or JForge depending on a challenge bit. In contrast to receiver compromise deniability, $\mathcal{A}_1$ generates the judge public key pair in addition to the receiver public key pair and $\mathcal{A}_2$ is given access to all secret keys. For an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and message franking scheme AMF we define the JCOMPDEN advantage of $\mathcal{A}$ against AMF as

$$\mathbf{Adv}^{\text{jden}}_{\mathsf{AMF},\mathcal{A}}(\lambda) = \left|\Pr\left[\text{JCOMPDEN}^{\mathcal{A},1}_{\mathsf{AMF}}(\lambda) = 1\right] - \Pr\left[\text{JCOMPDEN}^{\mathcal{A},0}_{\mathsf{AMF}}(\lambda) = 1\right]\right| .$$

Space of deniability definitions. Notice that our deniability definitions are implicitly parameterized by the combination of secrets keys given to the forger and the combination of secret keys given to the distinguisher, i.e., who is able to fool whom. In this work, we target three specific deniability definitions within this space that we believe have real-world significance. However, this is not the only set of meaningful deniability definitions that one might desire from a scheme. Consider the following two examples. First, our definitions give the distinguisher access to the sender's secret key which models deniability in the face of sender compromise. An alternative definition may dispense

with this goal in favor of an accountability notion, disavowability, in which a sender has the ability to cryptographically prove forged signatures were not created using their sender secret key, i.e., disavow forgeries. Second, our judge compromise deniability definition conflicts with strong authentication between sender and receiver — forgeries by the moderator cannot be detected by the receiver. Instead, a stronger unforgeability definition could be satisfied in which the judge's secret key alone is not sufficient to forge messages accepted by the receiver.

3.1.3 Construction

In this section, we present our construction for building an asymmetric message franking scheme. First, we give intuition for our approach by drawing connections to the literature on designated verifier signatures [JSI96, LWB05, KP05, LV04a]. Then, we describe our particular instantiation built using signatures of knowledge [Cam98] and detailed in Figure 3.5.

Intuition: AMF from Designated Verifiers

Designating the moderator as verifier. The tension between accountability and deniability arises from the desire for franking signatures to be forgeable (deniability) as well as verifiable by certain special parties, e.g. the moderator (accountability). This suggests *designated verifier* signatures [JSI96] as a natural starting point from which to build asymmetric message franking. The sender would designate the moderator as a verifier for a signature of the message.

A designated verifier signature or, more generally, a designated verifier proof system allows a prover to provide a proof of a statement that convinces a designated verifier but no one else. The designated verifier can efficiently forge the proof such that the forged proof is indistinguishable from a real proof even with access to the designated verifier's secret key. This security property, known as non-transferability, ensures there are two possible parties that could have created the signature, the alleged sender or the (compromised) moderator. It matches closely to receiver compromise deniability and judge compromise deniability for AMFs which extends the idea of non-transferability to relationships between three parties.

Universal deniability from strong designated verifiers. To expand the set of possible forgers to any

non-participating party, i.e. universal deniability, we additionally make use of a strong deniability property of *strong designated verifier* signatures [JSI96, SKM03, HYWS11]. This property allows anyone to forge a signature between two parties such that the resulting forgery is indistinguishable from real signatures to anyone without secret key access. Without care, universal deniability poses a problem for accountability. Consider a franking signature that consists of the sender creating a strong designated verifier signature for the moderator. A sender can send an abusive message and sign with a universal forgery. If the recipient of the message attempts to report to the moderator, the moderator will not be convinced the message was sent by the sender. This violates sender binding.

Chaining designated verifier proofs. To achieve sender binding, the receiver must have some way of verifying whether messages it receives are reportable to the moderator. Specifically, the receiver must be able to verify the sender's strong designated verifier signature for the moderator is well-formed and not a forgery. This leads us to the final step: the sender can attach a strong designated verifier proof [DFN06, CC18] for the receiver *proving* that the strong designated verifier signature for the moderator is well-formed. By using a strong designated verifier proof for this step, the deniability goals are preserved.

The challenge in building AMFs with this approach is in instantiating schemes such that the signing algorithm of the strong designated verifier signature falls into a language compatible with the strong designated verifier proof system. Existing strong designated verifier signatures [SKM03, HYWS11, JSI96] do not appear to have this desired structure-preserving property [AFG+10] that would lend to using efficient proof systems. Strong designated verifier proof systems for arbitrary languages would likely be prohibitively expensive for low latency messaging. It is worth noting that this approach is also conceptually similar to multi-designated verifier signatures, but in our case, the different parties have different forging powers [LV04b, DHM+20]. We next turn to building practical AMFs.

AMF from Signatures of Knowledge

While we do not build off the abstraction of designated verifiers, our construction is modeled off the intuition that an AMF can be composed of a strong designated verifier proof to the receiver of the well-formedness of a strong designated signature to the moderator. Our construction is

Algorithm	Security notion	How to prove first clause? $(pk_s = g^t \vee J = g^u)$	How to prove second clause? $\big((J = (pk_j)^v \wedge E_J = g^v) \vee R = g^w\big)$	Verify?	Judge?
Frank	Correctness	$\alpha \leftarrow_\$ \mathbb{Z}_p;\ J \leftarrow (pk_j)^\alpha;\ t = sk_s$	$\beta \leftarrow_\$ \mathbb{Z}_p;\ R \leftarrow (pk_r)^\beta;\ v = \alpha$	✓	✓
Forge	Universal deniability	$\gamma \leftarrow_\$ \mathbb{Z}_p;\ J \leftarrow g^\gamma;\ u = \gamma$	$\delta \leftarrow_\$ \mathbb{Z}_p;\ R \leftarrow g^\delta;\ w = \delta$	×	×
RForge	Receiver compromise deniability	$\gamma \leftarrow_\$ \mathbb{Z}_p;\ J \leftarrow g^\gamma;\ u = \gamma$	$\beta \leftarrow_\$ \mathbb{Z}_p;\ R \leftarrow (pk_r)^\beta;\ w = \beta \cdot sk_r$	✓	×
JForge	Judge compromise deniability	$\alpha \leftarrow_\$ \mathbb{Z}_p;\ J \leftarrow (pk_j)^\alpha;\ u = \alpha \cdot sk_j$	$\beta \leftarrow_\$ \mathbb{Z}_p;\ R \leftarrow (pk_r)^\beta;\ v = \alpha$	✓	✓

Figure 3.4: Summary of how AMF signing and forging algorithms construct signatures. The rightmost columns indicate with a checkmark (✓) which verification algorithms accept that signature and with a cross (×) which will reject that signature.

inspired by the strong designated verifier signature scheme of Huang et al. built using signatures of knowledge [HYWS11], which we modify to allow for proofs of well-formedness.

Our construction can be based on any suitable cyclic group. In the following we let $\mathbb{G}$ be a group, let p be its order, and g be a generator for $\mathbb{G}$. Secret keys are uniformly chosen from $\mathsf{SK} = \mathbb{Z}_p$, and public keys are set to be $pk \leftarrow g^{sk}$. We denote this key generation as Keygen. Note that it is easy to check the well-formedness of such keys.

Overview of construction. Consider the strong designated verifier signature (between sender and moderator) derived as a signature of knowledge from the following relation:

$$\mathsf{R}_{\mathsf{SDVS}} = \big\{\big((t, u), (g, pk_s, J)\big)\ :\ pk_s = g^t \vee J = g^u\big\},$$

in which an honest sender will construct Diffie-Hellman value $J = (pk_j)^\alpha$ for random choice of $\alpha \leftarrow_\$ \mathbb{Z}_p$, and send ephemeral value $E_J = g^\alpha$ along with the $\mathsf{R}_{\mathsf{SDVS}}$ signature proof, where pk_s and pk_j are the public keys of the sender and moderator, respectively. If J is indeed constructed in this manner, $J = g^u = g^{\alpha \cdot sk_j}$, then knowledge of u cannot be proved by anyone who does not know the moderator's secret key sk_j. This means a moderator that receives a valid signature and well-formed J will be convinced that the signature comes from a sender with knowledge of $t = sk_s$.

On the other hand, anyone can create a valid signature of $\mathsf{R}_{\mathsf{SDVS}}$ by using a malformed J set as a random group element, $J = g^\gamma$ for $\gamma \leftarrow_\$ \mathbb{Z}_p$, proving knowledge of $u = \gamma$, and sending $E_J = g^\alpha$ for independent $\alpha \leftarrow_\$ \mathbb{Z}_p$. Importantly, only the moderator has the ability to distinguish between well-formed and malformed J, by using the secret key sk_j to check whether (pk_j, E_J, J) forms a valid Diffie-Hellman triple ($J \stackrel{?}{=} E_J^{sk_j}$). This means that anyone can create a forged signature that is indistinguishable from a valid sender signature to everyone but the moderator.

$$\mathsf{R}_{\mathsf{AMF}} = \left\{ \big((t, u, v, w), (g, pk_s, pk_r, pk_j, J, R, E_J)\big) \,:\, \big(pk_s = g^t \vee J = g^u\big) \wedge \big((J = (pk_j)^v \wedge E_J = g^v) \vee R = g^w\big) \right\}$$

$\underline{\mathsf{AMF.Frank}(sk_s, pk_r, pk_j, m)}$

$(\alpha, \beta) \leftarrow\!\!\$\, (\mathbb{Z}_p)^2$
$J \leftarrow (pk_j)^\alpha$
$R \leftarrow (pk_r)^\beta$
$E_J \leftarrow g^\alpha$
$E_R \leftarrow g^\beta$
$w \leftarrow (sk_s, \perp, \alpha, \perp)$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
$\pi \leftarrow\!\!\$\, \Pi.\mathsf{SignProve}(m, w, x)$
Return (π, J, R, E_J, E_R)

$\underline{\mathsf{AMF.Forge}(pk_s, pk_r, pk_j, m)}$

$(\alpha, \beta, \gamma, \delta) \leftarrow\!\!\$\, (\mathbb{Z}_p)^4$
$J \leftarrow g^\gamma$
$R \leftarrow g^\delta$
$E_J \leftarrow g^\alpha$
$E_R \leftarrow g^\beta$
$w \leftarrow (\perp, \gamma, \perp, \delta)$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
$\pi \leftarrow\!\!\$\, \Pi.\mathsf{SignProve}(m, w, x)$
Return (π, J, R, E_J, E_R)

$\underline{\mathsf{AMF.RForge}(pk_s, sk_r, pk_j, m)}$

$(\alpha, \beta, \gamma) \leftarrow\!\!\$\, (\mathbb{Z}_p)^3$
$J \leftarrow g^\gamma$
$R \leftarrow (pk_r)^\beta$
$E_J \leftarrow g^\alpha$
$E_R \leftarrow g^\beta$
$w = (\perp, \gamma, \perp, \beta \cdot sk_r)$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
$\pi \leftarrow\!\!\$\, \Pi.\mathsf{SignProve}(m, w, x)$
Return (π, J, R, E_J, E_R)

$\underline{\mathsf{AMF.JForge}(pk_s, pk_r, sk_j, m)}$

$(\alpha, \beta) \leftarrow\!\!\$\, (\mathbb{Z}_p)^2$
$J \leftarrow (pk_j)^\alpha$
$R \leftarrow (pk_r)^\beta$
$E_J \leftarrow g^\alpha$
$E_R \leftarrow g^\beta$
$w \leftarrow (\perp, \alpha \cdot sk_j, \alpha, \perp)$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
$\pi \leftarrow\!\!\$\, \Pi.\mathsf{SignProve}(m, w, x)$
Return (π, J, R, E_J, E_R)

$\underline{\mathsf{AMF.Setup}(\lambda)}$

Return $\mathsf{GGen}(\lambda)$

$\underline{\mathsf{AMF.Keygen}(pp)}$

$(p, \mathbb{G}, g) \leftarrow pp$
$r \leftarrow\!\!\$\, \mathbb{Z}_p$
Return (g^r, r)

$\underline{\mathsf{AMF.Verify}(pk_s, sk_r, pk_j, m, \sigma)}$

$(\pi, J, R, E_J, E_R) \leftarrow \sigma$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
Return $\bigwedge \begin{pmatrix} R = E_R^{sk_r} \\ \Pi.\mathsf{Ver}(m, \pi, x) \end{pmatrix}$

$\underline{\mathsf{Judge}(pk_s, pk_r, sk_j, m, \sigma)}$

$(\pi, J, R, E_J, E_R) \leftarrow \sigma$
$x \leftarrow (g, pk_s, pk_r, pk_j, J, R, E_J)$
Return $\bigwedge \begin{pmatrix} b_1 \leftarrow J = E_J^{sk_j} \\ \Pi.\mathsf{Ver}(m, \pi, x) \end{pmatrix}$

Figure 3.5: Algorithms for our deniable AMF scheme $\mathsf{AMF}[\mathsf{GGen}, \Pi]$. The protocol is parameterized by a group generation algorithm GGen and a signature of knowledge Π for relation $\mathsf{R}_{\mathsf{AMF}}$.

Following the intuition from the previous section, to achieve accountability, the sender must prove to the receiver that the strong designated verifier signature for the moderator is well-formed. This corresponds to proving that J is well-formed, i.e., (pk_j, E_J, J) form a Diffie-Hellman triple. Putting it together, our final AMF construction is the signature of knowledge derived from the following relation:

$$\mathsf{R}_{\mathsf{AMF}} = \Big\{ \big((t, u, v, w), (g, pk_s, pk_r, pk_j, J, R, E_J)\big) : \\ \big(pk_s = g^t \vee J = g^u\big) \wedge \big((J = (pk_j)^v \wedge E_J = g^v) \vee R = g^w\big) \Big\}.$$

An honest sender constructs $J = (pk_j)^\alpha$ and $R = (pk_r)^\beta$ for $(\alpha, \beta) \leftarrow\!\!\$\, (\mathbb{Z}_p)^2$, and sends ephemeral values $(E_J = g^\alpha, E_R = g^\beta)$ along with the $\mathsf{R}_{\mathsf{AMF}}$ signature, where pk_r is the public key of the receiver. The first conjunction clause represents the strong designated verifier signature to the moderator and the second conjunction clause represents the strong designated proof to the receiver that the first

clause is constructed properly. Forgeries for universal deniability are created with malformed J and R, forgeries for receiver compromise deniability with malformed J, and forgeries for judge compromise deniability do not use any malformed elements. Lastly, the receiver's public key is added to the statement even though it does not appear in the proof relation, so that it is bound by the Fiat-Shamir hash challenge. This prevents certain types of identity misbinding attacks. A complete summary of how different signatures and forgeries are proved is given in Figure 3.4 and our full construction is detailed in pseudocode in Figure 3.5. The resulting primitive can be thought of as a sort of multi-designated verifier signature [DHM$^+$20] where the designated verifiers have differing hierarchical deniability properties.

3.1.4 Security Analysis

We now explore the security of our deniable AMF scheme, arguing it achieves the accountability and deniability properties detailed in Section 3.1.2. We treat each set of properties in turn.

Accountability. As we discussed in the last section, the accountability properties intuitively follow from the underlying signature of knowledge's soundness properties: demonstrating forgeries that fool the recipient (unforgeability or sender binding) or the judge (receiver binding) implies the ability to generate a proof without a witness. However, it is not clear how to modularly define a suitably strong knowledge soundness property of the signature of knowledge underlying our construction. Our analyses therefore take a different tack, reducing to the soundness properties of the underlying proof of knowledge protocol.

We discuss receiver binding, which shares the same high level strategy as sender binding. Our strategy is to show a winning adversary $\mathcal{A}$ breaks the one-wayness of the witness-statement relation $\mathsf{R}_{\mathsf{AMF}}$, which we can use to build a discrete log adversary $\mathcal{B}$ extracting secret keys from the witness. The approach of the proof uses some techniques related to the proof of existential unforgeability under chosen message attack (EUF-CMA) for Fiat-Shamir-derived signatures (c.f., [BS17]), but the need of $\mathcal{B}$ to simulate $\mathcal{A}$'s oracle queries requires a more nuanced analysis. In fact performing this simulation leads us to make an additional knowledge-of-exponent assumption (KEA) assumption [BP04] about $\mathbb{G}$. The full theorem for receiver binding is given below along with a proof sketch; the full proofs and theorem statement for sender binding and unforgeability are deferred to the full version [TGL$^+$19b].

Theorem 1. *Let* AMF *be the asymmetric message franking scheme defined in Figure 3.5 where the signature of knowledge is derived from the proof of representation Sigma protocol for* $\mathsf{R}_{\mathsf{AMF}}$ *using the Fiat-Shamir heuristic as described in Section 2.4 with hash function* H*. If* H *is modeled as a random oracle, for any* RBIND *adversary* $\mathcal{A}_{\text{rbind}}$ *making at most* q_{f} *franking oracle queries,* q_{j} *judge oracle queries, and* q_{ro} *random oracle queries, we give adversaries* $\mathcal{A}_{\text{dl}}$ *and* $\mathcal{A}_{\text{kea}}$ *such that*

$$\mathbf{Adv}^{\text{rbind}}_{\mathsf{AMF},\mathcal{A}_{\text{rbind}}}(\lambda) \leq \frac{q_{\mathsf{f}}(q_{\mathsf{f}}+q_{\mathsf{ro}}+1)}{p^4} + (q_{\mathsf{j}}+1)\cdot \mathbf{Adv}^{\text{kea}}_{\mathsf{GGen},\mathcal{A}_{\text{kea}},\mathsf{X}}(\lambda) + \frac{q_{\mathsf{ro}}+1}{p} + \sqrt{2(q_{\mathsf{ro}}+1)\cdot \mathbf{Adv}^{\text{dl}}_{\mathsf{GGen},\mathcal{A}_{\text{dl}}}(\lambda)}$$

where p *is the order of* $\mathbb{G}$ *output by* $\mathsf{GGen}(\lambda)$*. If* $\mathcal{A}$ *runs in time* T *and* KEA *extractor* X *runs in time* t_{X}*, then* $\mathcal{A}_{\text{dl}}$ *runs in time* $T' \approx 2T + 2(q_{\mathsf{j}}+1)\cdot t_{\mathsf{X}}$ *and* $\mathcal{A}_{\text{kea}}$ *runs in time* $T' \approx T$*.*

Proof sketch: Our proof proceeds via a sequence of games. The first set of game hops show how the game can be modified to answer $\mathcal{A}_{\text{rbind}}$'s franking queries without using the sender's secret key sk_s. Similarly to proving non-interactive zero knowledge for Fiat-Shamir-derived proofs [BS17, Theorem 20.3], this is done by programming the random oracle H to be consistent with the commitments used in the underlying Sigma protocol. This programming fails if a (randomly chosen) commitment collides with a value previously used as input to the random oracle. This happens with low probability as commitments are four uniformly chosen group elements. The birthday-bound term accounts for the probability of such a commitment collision.

The second set of game hops handles simulating the judge oracle without the judge's secret key. To do so we argue that one can simulate the queries using KEA extractors and, if that fails, we can build an adversary $\mathcal{A}_{\text{kea}}$ that violates the KEA. In fact this step uses a hybrid argument which gradually replaces each oracle call with an extractor-utilizing simulation of the check. This accounts for the second term of the theorem's advantage bound.

Finally we are in a game now in which the only use of the judge and sender secret keys is to define the public keys. We use a rewinding lemma [BS17, Lemma 19.2]. If $\mathcal{A}_{\text{rbind}}$ succeeds at forging in one execution against a particular message, we can rerun $\mathcal{A}_{\text{rbind}}$ ("rewind" it) with a different random oracle output for that message. The rewinding lemma lower bounds the probability that $\mathcal{A}_{\text{rbind}}$ succeeds twice in a row by the probability that it succeeds once. In turn, if one can forge twice

with different hash outputs, this allows extracting a witness from the Fiat-Shamir proof of knowledge. The last step involves a case analysis over the relation $\mathsf{R}_{\mathsf{AMF}}$ to show that extracting a witness implies learning sk_s or sk_j, which we use to build our desired discrete log adversary $\mathcal{A}_{\text{dl}}$. A subtlety in this final step is that extracting a witness implies learning $u = sk_j \cdot \alpha$, but not sk_j directly. We use a KEA extractor again to extract α, and thus complete the proof. This accounts for the final two terms of the advantage relation.

Deniability. Intuitively, the deniability properties fall out of the non-interactive zero knowledge property of the signature proofs of knowledge. Our signature proof of knowledge is carefully designed so that a variety of different witnesses can satisfy the statement relation $\mathsf{R}_{\mathsf{AMF}}$ (as laid out in Figure 3.4). This allows forgers to create signatures that can only be caught by checking well-formedness of the statement using secret keys.

In more detail, the deniability proofs all follow the same outline. First notice that there are two high level differences between the frank algorithm and the forge algorithms: (1) the witnesses used to prove the statement are different, and (2) how the elements of the statement are formed is different. Different witnesses are handled by using the zero-knowledge property of the signature proof to switch between witnesses by hopping to a simulated proof and back. In fact, for judge compromise deniability, witness indistinguishability [FS90] is all that is needed since elements of the statement are well-formed and identical in Frank and JForge. Extra care needs to be taken for Forge and RForge, since some elements of the statement are malformed. Well-formed means, for example, that J is constructed as $J \leftarrow (pk_j)^\alpha$ forming a Diffie-Hellman triple, $(pk_j = g^{sk_j}, E_J = g^\alpha, J = g^{\alpha \cdot sk_j})$. While malformed means $J \leftarrow g^\gamma$ is constructed as a random group element. In RForge, J is malformed, while in Forge both J and R are malformed. This leads to an additional DDH term to bound the advantage of an adversary in distinguishing between each well-formed and malformed statement elements.

The theorem statement for universal deniability is given below. The first term of the advantage comes from hopping between two witnesses through a simulator. The second term of the advantage comes from a decisional Diffie-Hellman hop for each of the two malformed elements of Forge.

Theorem 2. *Let* AMF *be the asymmetric message franking scheme defined in Figure 3.5 where the signature of knowledge is derived from the proof of representation Sigma protocol* Π *for* $\mathsf{R}_{\mathsf{AMF}}$ *using the Fiat-Shamir heuristic as described in Section 2.4 with hash function* H*. For all simulators* S *for* Π*,*

for any UDEN *adversary* $\mathcal{A}_{\text{uden}}$*, we give adversaries* $\mathcal{A}_{\text{zk}}$ *and* $\mathcal{A}_{\text{ddh}}$ *such that*

$$\mathbf{Adv}^{\text{uden}}_{\mathsf{AMF},\mathcal{A}_{\text{uden}}}(\lambda) \leq 2 \cdot \mathbf{Adv}^{\text{zk}}_{\Pi,\mathsf{R}_{\mathsf{AMF}},\mathsf{S},\mathcal{A}_{\text{zk}}}(\lambda) + 2 \cdot \mathbf{Adv}^{\text{ddh}}_{\mathsf{GGen},\mathcal{A}_{\text{ddh}}}(\lambda) \,.$$

where if $\mathcal{A}_{\text{uden}}$ *runs in time* T *and makes at most* q_{f} *queries to the frank oracle, then* $\mathcal{A}_{\text{zk}}$ *and* $\mathcal{A}_{\text{ddh}}$ *run in time* $T' \approx T$ *and* $\mathcal{A}_{\text{zk}}$ *makes at most* q_{f} *queries to its proof oracle.*

The advantage terms for receiver compromise deniability and judge compromise deniability follow a similar structure. The full proofs for all deniability properties are deferred to the full version [TGL+19b].

3.2 Message Tracing

Another setting in which traditional SMF fails is in combatting misinformation campaigns. Such campaigns are being increasingly hosted on messaging platforms in which parties send messages with misleading or false information, and encourage them to be forwarded by the recipient. Such campaigns can have serious consequences, contributing political instability and inciting violence.

In unencrypted contexts, such as Twitter or the Facebook news feed, platforms have started to combat misinformation campaigns with content moderation, tracing harmful messages sent through their network and intervening as deemed appropriate, e.g., by banning the "factory" accounts that are injecting such content into the network. But E2E encryption complicates moderation because the platform never observes plaintext content. Message franking [Fac17, GLR17, DGRW18, TGL+19b] (both symmetric and asymmetric) allows cryptographically verified content moderation, but current techniques only reveal the sender of a received message and do not by themselves help identify the source of a forwarded message. Platforms have so far relied solely on crude techniques like limiting the number of people to which any message can be forwarded.

We explore a new tool for content moderation in E2E encrypted messaging: traceback. At a high level, tracing should allow users to report an abusive message along with supporting cryptographic key material to the messaging platform. Using this material, the platform can recover a cryptographically verifiable trace of the message, revealing the source and how the message was forwarded between users.

We design two traceback schemes. The first, called path traceback, uses lightweight symmetric encryption techniques to add specially constructed tracing tags to ciphertexts. A tracing tag is an

encrypted pointer to either the prior message, in the case of a forward, or a distinguished symbol in the case of a freshly authored message. The ability to decrypt the tracing tag is secret-shared across the platform and the recipient, so that only when a recipient reports the message can the platform decrypt the tracing tag and reveal information about forwards. By carefully ensuring that subsequent forwards' tracing tags form an encrypted linked list, the platform can, given an abuse report, trace back to the source of the message content.

Our second scheme is called tree traceback. It extends path traceback to additionally allow tracing forward from the original source to recover all the recipients of forwards of the message content. Tree traceback could be useful to platform operators when mitigating and cleaning up after abuse, since it allows, for example, identifying the victims of misinformation campaigns. Achieving it is more complicated, however, in particular because we want to build an encrypted tree data structure incrementally, allowing tracing to connect all forwards of a message, including ones that will happen in the future. We nevertheless show how to achieve it.

Our two schemes are practical to deploy. By design, they work with arbitrary E2E encryption systems. They utilize only fast symmetric encryption and cryptographic hashing, and add a small number of bytes to each encrypted message. The schemes do require that the platform store a short trace tag (<100 bytes) for each encrypted message sent, which is nevertheless practical even for high-volume messaging systems. We implement a prototype of both of our schemes, report on initial performance evaluation, and detail how traceback can be easily integrated into existing E2E encryption protocols such as Signal.

Related work. A number of works, including [CMS96, FTY96, KV02] have considered the problem of tracing payments in electronic cash systems. This is a conceptually related problem. However, the techniques are not directly applicable. First, these systems do not deal with binding the content of a message to the trace. Second most systems assume far more interaction with a central party (e.g., a bank) than is allowed in our setting.

Another line of work [SWKA00, SP01, Sno01] considered IP packet traceback. But the approaches taken here are probabilistic. The point is not to trace an individual packet, but to get a trace of a stream of packets. As such packets are probabilistically marked [SWKA00] or kept in a Bloom filter [Sno01]. They cannot be used to reliably trace an individual message.

Lastly, another line of work explores traitor tracing mechanisms [CFN94, BF99] for identifying who within a group leaked a particular piece of content or a key to an outside party. These schemes are not directly applicable for a few reasons. First, the goals of message tracing are not to trace who leaked/reported the message to the platform (out of the forwarding chain "group"). Second, message tracing has very specific restrictions on who is able to perform tracing. For example, by user trace confidentiality, even members of the forwarding chain "group" should not be able to learn the message trace.

3.2.1 Traceback Setting in Messaging

We consider an E2E encrypted message setting, in which a *platform* helps *users* send encrypted messages. A primary goal of such messaging services is confidentiality of user messages. While some systems [vdHLZZ15, TGL+17, WCFJ12] also target metadata privacy, i.e., obfuscating from the platform who is the sender or recipient (or both), we restrict attention to systems such as Facebook secret messenger and Whatsapp that reveal such metadata to platforms.

Messaging clients allow *forwarding* of encrypted messages. While forwarding is beneficial to legitimate users, it has also been subject to abuse by users spreading malicious content. In this chapter, we show how to augment encrypted messaging systems to allow users to *report* a malicious message. The platform can, given this report, trace the path a message took as it was forwarded across the network of users. This enables new moderation approaches, as we discuss more at the end of this section. We first discuss our confidentiality and accountability goals in more detail.

Confidentiality goals. We want to support traceback while minimizing impact on E2E confidentiality guarantees. Messages and forwarding behavior (whether an encrypted message is a forward, and from whom) should be confidential even from the platform, except in the case that the message was reported. For non-reported messages, neither the platform nor users should ever learn anything new due to the tracing functionality. In particular, we want:

- *Trace confidentiality for users*: Users should not learn any information about message paths beyond their local view of receiving and sending messages.
- *Pre-report trace confidentiality for platform*: Before a report, the platform should not learn any additional information about the message path beyond communication metadata (e.g., receiver,

sender, timing, message length).

- *Post-report trace confidentiality for platform*: After a report, the platform should learn the message trace and nothing more.

These confidentiality goals emanate from our intention to hew closely to the existing behavior and privacy offered by deployed E2E messaging systems. First, we have chosen to ensure that tracing does not reveal to even a malicious user if they are receiving a forwarded message or a fresh one. Here, deployed systems take different approaches and we chose the approach that maximizes compatibility. Whatsapp reveals to the recipient that a message is a forward, but does not reveal from whom. Signal, in contrast, does not identify forwarded messages. By providing forwarding privacy, our tracing schemes work in both contexts.

Second, we have chosen to explicitly reveal messaging metadata. This means the platform can tell that A communicated with B who then communicated with C. Our second choice is to conceal from the platform whether B sent a fresh message to C or forwarded the message they got from A. Together these choices preserve the general property of E2E encrypted messaging systems such as WhatsApp and iMessage: the platform learns who messages who but nothing about the content of the message.

Traffic analysis by a (compromised or otherwise malicious) platform may allow inferring forwarding behavior. For example, if the platform observes a single 125 byte inbound message to some user followed shortly after by 20 outbound messages of size 125 bytes, it is likely a sequence of forwards. We will not attempt to prevent such traffic analysis, which would require expensive padding and timing obfuscations.

Finally we note that our schemes will not interfere with cryptographic deniability. In the messaging setting, this refers to the idea that recipients should not be able to provide to a third party cryptographic evidence proving that the sender sent a particular plaintext. For example, providing a recipient with a digital signature of a plaintext using the sender's private key would violate our goals. Deniability was explicitly sought by Facebook's message franking system [Fac17], and by using symmetric primitives we will achieve whatever level of deniability is offered by the underlying E2E encryption mechanisms.

Accountability goals. In this paper, we propose two different types of traceback. The simplest form

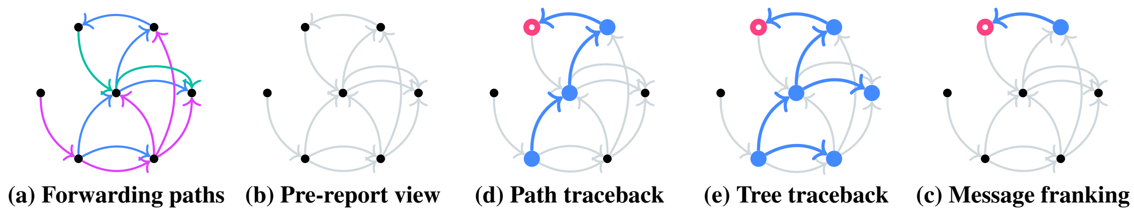

Figure 3.6: Example of message forwards and what different traceback mechanisms report. Nodes are users and edges are message sends, with forwarding paths denoted in the same color. Subfigures (c), (d), and (e) depict what is revealed to the platform when the blue message is reported by the user indicated by the red circle.

is *path traceback* which allows a platform to trace a reported message back to its origin, identifying every forward along the path. This allows the originator to be held accountable for a message that is forwarded. Separately, we consider *tree traceback*, which allows messages to be traced both back to their origin and to identify all forwards of the message. This enables not only the identification of the sender, but also of recipients. It may be useful for helping notify users about malicious content, or blocking further forwards of the message. A particular complication for tree traceback is that the platform should be able to trace all forwards including those that occur after the report is made. A visualization of the different traceback policies are shown in Figure 3.6. Shown also there is what message franking [Fac17, GLR17, DGRW18] supports in terms of accountability. Our path and tree traceback can be seen as generalizations of message franking.

As was done previously with message franking, we want to ensure various accountability properties even in the face of malicious users. But our setting is more complicated because tracing requires reasoning about multiple messages sent, as opposed to a single one. Intuitively, we require that no colluding set of adversarial users can frame an honest user as having performed some action (sending, receiving, or forwarding a message with some particular plaintext content) that they did not, in fact, perform. This implies, for example, the following accountability goals:

- *Trace unforgeability*: No group of colluding users can generate a report implicating an honest user as having performed an action they did not, in fact, perform.
- *Sender binding*: No user can author a message that cannot be traced back to them.

Note that trace unforgeability can be seen as a generalization of the receiver binding property targeted by message franking.

There are some limits to the level of accountability that we target (and that our eventual schemes

will provide). In particular, we allow for malicious users to partition the traceback. In these "partition attacks", a malicious user can split a forwarding chain so as to make themselves appear as the end of one side of the split and the source of the other side. This seems fundamentally unavoidable. Consider that a malicious user can bypass the cryptographic forwarding mechanism and emulate a forward by resending a message via a copy-paste of the received plaintext. In theory, the client software could be modified to flag such copy-paste forwards by comparing incoming and outgoing plaintexts (after decryption and before encryption, respectively). Detection is not straightforward, particularly if one wants it to be robust to small changes in the message. Moreover, this kind of defense relies upon the integrity of the client software, as an adversary using a compromised client can avoid client-side detection logic and directly submit encrypted content to the platform. Future work might target prevention of partition attacks despite compromised clients through the use of more heavyweight tools such as trusted hardware and expensive zero-knowledge proofs.

For the purposes of this work, we require only that a malicious user will be left holding "both ends" of a partitioned chain. That is, there will be a trace including the malicious user as having received the original message and a (possibly different) trace including the malicious user as having sent the message. In the end, this means that we will guarantee messages will be traceable either to the original source *or* to a malicious user that partitioned the chain.

The effectiveness of tracing. A final question arises: if it is possible for a user to bypass our proposed tracing protections via a hacked client, is tracing effective? We believe so. Abuse mitigation techniques need not be perfect to be effective, and even just reducing the amount of abuse is worthwhile. There is empirical evidence that protections that can be bypassed even by simple copy-paste behavior can still be effective. WhatsApp ran an experiment on the effectiveness of limiting the number of forwards a user could make of a given message. They found that this was effective despite the fact that the user could circumvent the limit after they reached it by simply copy and pasting the message. As a result, this forward limitation is now deployed globally.

3.2.2 Syntax and Security Notions

In this section, we present the syntax and semantics we will use to describe message tracing schemes. We assume users $u_1, u_2, \ldots, u_n$ each represented by a unique identifier taken from some set $\mathcal{U}$. For

convenience later, we assume a distinguished user identifier $\perp$ that no real user can use. We make minimal assumptions on user identities, assuming only that they are unique and that the platform can authenticate them. In practice one will use the identifiers already used in E2E messaging systems.

We use the term *message* to refer to the sending of some plaintext from one user to another at some point in time. Multiple messages may have the same plaintext (e.g., because someone forwards message plaintext or sends the same message to multiple people). The distinction between message and plaintext will be particularly critical in our discussion of tracing scheme accountability properties.

Our formalization of a message tracing scheme is decoupled from the underlying end-to-end encryption. This leads to a modular and flexible deployment path in that any message tracing scheme can be used in conjunction with any (non-metadata-private) end-to-end encryption algorithm.

A message tracing scheme $\mathsf{MT} = (\mathsf{Setup}, \mathsf{InitU}, \mathsf{InitP}, \mathsf{NewMsg}, \mathsf{TagGen}, \mathsf{RecMsg}, \mathsf{Process}, \mathsf{Trace})$ is a tuple of algorithms. The setup algorithm produces public parameters pp given a security parameter input λ. The public parameters are input to all following algorithms; here we denote as a superscript, but we will omit if clear from context. The initialization algorithms produces initial states for users and the platform. The following three algorithms are called by users when sending and receiving messages. The last two algorithms are for the platform to trace messages given the proper user-provided key material.

- $tmd \leftarrow\!\!\$\ \mathsf{NewMsg}^{pp}(u, pt : st_{\mathrm{u}})$: The randomized message authorship algorithm takes in a user, a message plaintext, and outputs trace metadata to be associated with this particular authored message instance.
- $(k, tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}^{pp}(u_s, u_r, pt, tmd : st_{\mathrm{u}})$: The randomized tag generation algorithm takes in the sender u_s and recipient u_r identities, a message plaintext, and trace metadata. The algorithm outputs a sender trace tag tt_s and tracing key k. The tracing key is to be included with the plaintext in the end-to-end encrypted ciphertext which is sent along with the sender trace tag to the recipient over the platform.
- $tmd \leftarrow \mathsf{RecMsg}^{pp}(k, u_s, u_r, pt, tt_r : st_{\mathrm{u}})$: The tag receive algorithm takes in a key, the sender u_s and recipient u_r identities, a message plaintext, and a recipient trace tag, then outputs trace metadata that cryptographically identifies the received message. The algorithm may return an error symbol $\perp$ (e.g., in the case tt_r is malformed).

- $((mid, tt_p), tt_r) \leftarrow\!\$\ \mathsf{Process}^{pp}(u_s, u_r, tt_s : st_\mathrm{p})$: The processing algorithm is run by the platform and takes in the platform state, the sender and receiver identities, and a sender trace tag. It outputs a recipient trace tag tt_r to deliver to the recipient as well as a message identifier mid and a platform trace tag tt_p. In our schemes, the platform updates st_p, which is a simple key-value store, to include (mid, tt_p).
- $tr \leftarrow \mathsf{Trace}^{pp}(u, pt, tmd : st_\mathrm{p})$: A user can report a received message by sending the plaintext pt and trace metadata for the message to the platform. The tracing algorithm is run by the platform and takes in the platform state, the reporting user identity, the message plaintext, and trace metadata. It then returns a trace tr of the reported message instance, the detailed structure of which depends on the tracing goal. For path traceback it corresponds to a path with nodes labeled by users and edges labeled by message identifiers. For tree traceback, a similarly-labeled tree is returned.

Usage. The algorithms for tracing described above are designed to be decoupled from the end-to-end encryption algorithms used by the messaging platform. A typical message is sent in the following manner. First, the sender must specify the message they wish to send, i.e., whether it is a new message or a forward. In either case we want to associate some trace metadata to the message. If the user authors their own message, this metadata is created using NewMsg. Otherwise, RecMsg generates trace metadata for a previously received message that can be used when forwarding. To send a message, the sender generates a tracing key k and a sender trace tag tt_s using TagGen with the appropriate trace metadata. The sender encrypts the tracing key and message plaintext using the E2E encryption protocol, and sends the resulting ciphertext along with tt_s to the platform.

The platform processes tt_s using Process, updating its internal state to log a message identifier mid and associated platform trace tag tt_p. Note that tt_p does not necessarily equal tt_s. It also derives a recipient trace tag tt_r and sends the E2E ciphertext and tt_r to the recipient. The recipient decrypts the ciphertext to recover the tracing key k and plaintext, and then uses RecMsg to both verify the received trace tag and generate the trace metadata that can be used to forward the message in the future. The recipient may report a message to the platform by sending the message plaintext and associated trace metadata to the platform. The platform uses Trace with its internal state to learn a trace of the reported message instance.

Correctness. Informally, correctness dictates that trace tags created with honest calls to NewMsg, TagGen, and RecMsg and processed by an honest platform using $\mathsf{Process}$ should (1) not fail well-formedness verification in RecMsg, and (2) provide the correct trace with Trace when reported. Correctness is therefore context dependent, and we will discuss it more in subsequent sections. We just note that most of our schemes will not be perfectly correct, but rather be correct with all but negligible probability.

Confidentiality Notions

We start by formalizing notions of security capturing our confidentiality goals. Recall that our confidentiality goals include: (1) trace confidentiality from the platform, meaning the platform learns nothing about message contents or message history unless a report implicates that message, and (2) trace confidentiality from users, meaning a user learns nothing about the history of messages they receive. We therefore formalize two notions of confidentiality.

Our confidentiality definitions isolate what might leak from the output of a specific honest node, even for adversarially chosen keys, tracing information, and messages. This ensures confidentiality goals even in more complicated attack settings, as well, for example distinguishing between a sequence of forwards and a sequence of new messages being sent.

Platform trace confidentiality. For platform trace confidentiality, we propose a real-or-random definition for the platform view, i.e., the sender trace tag of a sent message. By using a real-or-random style definition, we capture both goals of platform traceback, hiding message content and hiding message history, within a single definition. In this game, given in Figure 3.7 (left), the adversary $\mathcal{A}$ plays the role of the platform and is provided with a tag generation challenge oracle that either returns the trace tag output from TagGen or a random string. The task of the adversary is to distinguish between the two where an adversary's advantage is defined as

$$\mathbf{Adv}^{\text{p-tr-conf}}_{\mathsf{MT},\mathcal{A}}(\lambda) = \left|\Pr\left[\mathrm{PT{\scriptsize R}C{\scriptsize ONF}}^{\mathcal{A},1}_{\mathsf{MT}}(\lambda) = 1\right] - \Pr\left[\mathrm{PT{\scriptsize R}C{\scriptsize ONF}}^{\mathcal{A},0}_{\mathsf{MT}}(\lambda) = 1\right]\right| .$$

User trace confidentiality. In user trace confidentiality, a real-or-random style definition will not work, as the recipient's view of tracing key, plaintext, and recipient trace tag have a related structure and must verify under RecMsg. Instead, we focus on the specific goal we aim to achieve under user

Game $\text{PTrConf}_{\mathsf{MT}}^{\mathcal{A},b}(\lambda)$	Game $\text{UTrConf}_{\mathsf{MT}}^{\mathcal{A},b}(\lambda)$
$pp \leftarrow\$ \mathsf{MT.Setup}(\lambda)$ For $u \in \mathcal{U}$ do $\quad st_u \leftarrow \mathsf{InitU}^{pp}$ $\quad T_u \leftarrow [\cdot]$; $ctr_u \leftarrow 0$ $b' \leftarrow\$ \mathcal{A}^{\text{NewMsg},\text{RecMsg},\text{Chal}}$ Return b'	$pp \leftarrow\$ \mathsf{MT.Setup}(\lambda)$ For $u \in \mathcal{U}$ do $\quad st_u \leftarrow \mathsf{InitU}^{pp}$ $\quad T_u \leftarrow [\cdot]$; $ctr_u \leftarrow 0$ $b' \leftarrow\$ \mathcal{A}^{\text{NewMsg},\text{RecMsg},\text{Chal}}$ Return b'
Oracle $\text{NewMsg}(u, pt)$ $tmd \leftarrow\$ \mathsf{NewMsg}^{pp}(u, pt : st_u)$ $T_u[ctr_u] \leftarrow (tmd, pt)$; $ctr_u \leftarrow ctr_u + 1$	Oracle $\text{NewMsg}(u, pt)$ $tmd \leftarrow\$ \mathsf{NewMsg}^{pp}(u, pt : st_u)$ $T_u[ctr_u] \leftarrow (tmd, pt)$; $ctr_u \leftarrow ctr_u + 1$
Oracle $\text{RecMsg}(u_s, u_r, pt, tt_r)$ $tmd \leftarrow \mathsf{RecMsg}^{pp}(k, u_s, u_r, pt, tt_r : st_{u_r})$ $T_u[ctr_u] \leftarrow (tmd, pt)$; $ctr_u \leftarrow ctr_u + 1$	Oracle $\text{RecMsg}(u_s, u_r, pt, tt_r)$ $tmd \leftarrow \mathsf{RecMsg}^{pp}(k, u_s, u_r, pt, tt_r : st_{u_r})$ $T_u[ctr_u] \leftarrow (tmd, pt)$; $ctr_u \leftarrow ctr_u + 1$
Oracle $\text{Chal}(u_s, u_r, ctr)$ Require $ctr \in T_{u_s}$; $(tmd, pt) \leftarrow T_{u_s}[ctr]$ $(k, tt_s^1) \leftarrow\$ \mathsf{TagGen}^{pp}(u_s, u_r, pt, tmd : st_{u_s})$ $tt_s^0 \leftarrow\$ \{0,1\}^{\mathsf{MT.ttlen}(\lambda)}$ Return tt_s^b	Oracle $\text{Chal}(st_{\mathrm{p}}, u_s, u_r, ctr)$ Require $ctr \in T_{u_s}$; $(tmd_0, pt) \leftarrow T_{u_s}[ctr]$ If $tmd_0 = \perp$ then return $\perp$ $tmd_1 \leftarrow\$ \mathsf{NewMsg}^{pp}(u_s, pt : st_{u_s})$ $(k', tt_s) \leftarrow\$ \mathsf{TagGen}^{pp}(u_s, u_r, pt, tmd_b : st_{u_s})$ $((mid, tt_p), tt_r') \leftarrow \mathsf{Process}^{pp}(u_s, u_r, tt_s : st_{\mathrm{p}})$ Return (k', tt_r')

Figure 3.7: Security games for platform trace confidentiality (left) and user trace confidentiality (right). Highlighted portions are included for schemes targeting tree traceback.

trace confidentiality, namely that message history is not revealed. We thus task the adversary with distinguishing between the result of an authored message and a forwarded message. The adversary gets to choose the plaintext and, in the case the challenge oracle is forwarding, the trace metadata for the message to be forwarded. The game pseudocode is given in Figure 3.7 (right). We define the distinguishing advantage of the adversary as:

$$\mathbf{Adv}_{\mathsf{MT},\mathcal{A}}^{\text{u-tr-conf}}(\lambda) = \left| \Pr\left[\text{UTrConf}_{\mathsf{MT}}^{\mathcal{A},1}(\lambda) = 1\right] - \Pr\left[\text{UTrConf}_{\mathsf{MT}}^{\mathcal{A},0}(\lambda) = 1\right] \right| .$$

Accountability Notions

Tracing should accurately identify the source of a message, but malicious users can always obfuscate from whom they've received a message. We therefore want tracing never to result in an honest user erroneously implicated in having taken an action (sent, forwarded, or received a message) they did not, in fact, perform.

To formalize accountability we use a game-based approach in which an adversary interacts with some number n of honest users. See Figure 3.8. The adversary can cause honest users to author and send (adversarially chosen) messages via two oracles $\textsc{NewMsg}$ and $\textsc{Send}$. The adversary can also pose as any number of malicious users, sending messages via a malicious send oracle $\textsc{SendMal}$. We assume $\mathcal{U}$ labels users by numbers, thus user i is honest if $i \in [1, n]$ and user i is malicious if $i \notin [1, n]$. In our exposition we often use variables $u_1, \ldots, u_n$ to refer to the honest users, and a_j for $j \notin [1, n]$ for a malicious user. The security experiment here assumes authenticated channels — the adversary cannot send messages as an honest user nor manipulate messages sent between the honest users and the platform. On the other hand, we give the adversary the power to observe trace tags generated by, or sent to, honest users, but they only see the tracing keys sent by honest parties to malicious users. Given the use of secure channels to send messages, it would seem sufficient to not reveal communications from honest parties to the platform and from the platform to honest parties, but giving the adversary this information only makes the security achieved stronger.

The adversary's goal is to generate a report that results in an *invalid* trace, one that indicates that an honest user took some action that they did not, in fact, take. Note that the adversary can either have an honest user or malicious user make a report. In the honest case, the adversary outputs a value $mctr^*$ indicating which message received by i^* is being reported, and in the malicious case the adversary directly outputs an opening k^*. The game loops over the reported trace (skipping the loop entirely if Trace output an error), and checks for each honest user implicated in the trace whether the reported trace matches an action they in fact performed. We do this via a set of predicates, corresponding to where in the trace the honest user appears, and whether they actually received and/or sent the indicated messages.

As a non-exhaustive list of example invalid traces ruled out by these predicates, consider the following scenarios, where for simplicity we use a single honest user ($n = 1$):

- *Message replacement*: Honest user u_1 sends a message mid with plaintext pt to a malicious user a_2, who then successfully reports the trace $pt^* : u_1 \xrightarrow{mid} a_2$ for some plaintext $pt^* \neq pt$. This frames the honest user as having sent the wrong plaintext. The only valid trace in this case is $pt : u_1 \xrightarrow{mid} a_2$.
- *Identity replacement*: Honest user u_1 sends a message mid with plaintext pt to a malicious user

$\underline{\text{Game } \textsc{TrUnf}^{\mathcal{A}}_{\mathsf{MT},n}(\lambda)}$

$pp \leftarrow\!\!\$\ \mathsf{MT.Setup}(\lambda)$
$st_{\mathrm{P}} \leftarrow \mathsf{InitP}^{pp}$
For $u \in [1,n]$ do
  $st_u \leftarrow \mathsf{InitU}^{pp}$
  $T_u \leftarrow [\cdot]\,;\ ctr_u \leftarrow 0$
$(u^*, pt^*, tmd^*, ctr) \leftarrow\!\!\$\ \mathcal{A}^{\textsc{NewMsg},\textsc{Send},\textsc{SendMal}}$
If $u^* \in [1,n]$ then $(tmd^*, pt^*, mid_s) \leftarrow T_{u^*}[ctr]$
$tr \leftarrow \mathsf{Trace}^{pp}(u^*, pt^*, tmd^* : st_{\mathrm{P}})$

$\left(\left[(tr_i, mid_{i,i+1})\right]_{i=1}^{\tau}, tr_\tau\right) \leftarrow tr$

Return
$$\left(\begin{array}{c} (u^* \in [1,n] \wedge \tau = 1) \\ \bigvee \\ \bigvee_{j=1}^{\tau} \left(tr_j \in [1,n] \wedge \left(\bigvee \left(\begin{array}{l} (j = 1 \wedge \neg \mathtt{WasSent}(tr_1, tr_2, mid_{1,2}, pt^*)) \\ (j = \tau \wedge \neg \mathtt{WasRec}(tr_{\tau-1}, tr_\tau, mid_{\tau-1,\tau}, pt^*)) \\ (1 < j < \tau \wedge \neg \mathtt{WasFwd}(tr_{i-1}, tr_i, mid_{i-1,i}, mid_{i,i+1}, pt^*)) \end{array} \right) \right) \right) \end{array} \right)$$

$(tr_0, clist_0) \leftarrow tr$

Return
$$\bigvee_{(mid_{0,1}, tr_1, clist_1) \in clist_0} \left(\begin{array}{l} (u^* \in [1,n] \wedge \neg \mathtt{WasSent}(tr_0, tr_1, mid_{0,1}, pt^*)) \\ \mathsf{check_tree}(tr_i, mid_{i,i+1}, tr_{i+1}, clist_{i+1}) \end{array} \right)$$

$\underline{\mathsf{check_tree}(tr)}$

$(tr_{i-1}, mid_{i-1,i}, tr_i, clist_i) \leftarrow tr$

Return
$$\bigvee \left(\begin{array}{l} clist_i = \emptyset \wedge tr_i \in [1,n] \wedge \neg \mathtt{WasRec}(tr_{i-1}, tr_i, mid_i, pt^*) \\ \bigvee_{(mid_{i,i+1}, tr_{i+1}, clist_{i+1}) \in clist_i} \left(\begin{array}{l} tr_i \in [1,n] \wedge \neg \mathtt{WasFwd}(tr_i, tr_{i+1}, mid_{i-1,i}, mid_{i,i+1}, pt^*) \\ \mathsf{check_tree}(tr_i, mid_{i,i+1}, tr_{i+1}, clist_{i+1}) \end{array} \right) \end{array} \right)$$

$\underline{\text{Oracle } \textsc{NewMsg}(u, pt)}$

Require $u \in [1,n]$
$tmd \leftarrow\!\!\$\ \mathsf{NewMsg}^{pp}(u, pt : st_u)$
$T_u[ctr_u] \leftarrow (tmd, pt, \mathsf{auth})$
$ctr_u \leftarrow ctr_u + 1$

$\underline{\text{Oracle } \textsc{Send}(u_s, u_r, ctr)}$

Require $u_s \in [1,n] \wedge ctr \in T_{u_s}$
$(tmd, pt, mid_s) \leftarrow T_{u_s}[ctr]$
$(k, tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}^{pp}(u_s, u_r, pt, tmd : st_{u_s})$
$((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Process}^{pp}(u_s, u_r, tt_s : st_{\mathrm{P}})$
$T_P[mid] \leftarrow tt_p$
If $mid_s = \mathsf{auth}$ then
  $\mathtt{WasSent}(u_s, u_r, mid, pt) \leftarrow \mathsf{true}$
Else $\mathtt{WasFwd}(i, j, mid_s, mid, pt_s) \leftarrow \mathsf{true}$
If $u_r \in [1,n]$ then
  $tmd \leftarrow \mathsf{RecMsg}^{pp}(k, u_s, u_r, pt, tt_r : st_{u_r})$
  $T_u[ctr_u] \leftarrow (tmd, pt, mid)$
  $ctr_u \leftarrow ctr_u + 1$
  $\mathtt{WasRec}(u_s, u_r, mid, pt) \leftarrow \mathsf{true}$
  Return tt_r
Return (tt_r, k)

$\underline{\text{Oracle } \textsc{SendMal}(u_s, u_r, pt, tt_s)}$

Require $u_s \notin [1,n]$
$((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Process}^{pp}(u_s, u_r, tt_s : st_{\mathrm{P}})$
$T_P[mid] \leftarrow tt_p$
If $u_r \in [1,n]$ then
  $tmd \leftarrow \mathsf{RecMsg}^{pp}(k, u_s, u_r, pt, tt_r : st_{u_r})$
  If $tmd \neq \bot$ then
    $T_u[ctr_u] \leftarrow (tmd, pt, mid)$
    $ctr_u \leftarrow ctr_u + 1$
    $\mathtt{WasRec}(u_s, u_r, mid, pt) \leftarrow \mathsf{true}$
Return tt_r

Figure 3.8: Trace unforgeability security game with adversary winning condition for path traceback highlighted in gray and for tree traceback highlighted in blue.

a_2, who then successfully reports the trace $pt : u_1 \xrightarrow{mid} a_3$ for some distinct user a_3. This frames the honest user as having sent the message to a different user. The only valid trace in this case is $pt : u_1 \xrightarrow{mid} a_2$.

- *Path suffix*: Malicious user a_2 sends a message mid_a with plaintext pt to the honest user u_1, and then u_1 forwards mid_a to another user a_3 in message mid_b. Then a_3 successfully reports the trace $pt : u_1 \xrightarrow{mid_b} a_3$. This frames the honest user as having originated a message that they instead forwarded from someone else. The valid traces that can be reported in this case are $pt : a_2 \xrightarrow{mid_a} u_1$ and $pt : a_2 \xrightarrow{mid_a} u_1 \xrightarrow{mid_b} a_2$.
- *Same-message, wrong path*: Two malicious users a_2, a_3 send messages mid_a, mid_b with the same plaintext pt to the honest user u_1. The honest user forwards the message mid_a from a_2 to a user a_4 in a message mid_c. Finally a_4 generates a report resulting in trace $pt : a_3 \xrightarrow{mid_b} u_1 \xrightarrow{mid_c} a_4$. This frames the honest user as having forwarded a different message, despite the plaintext being the same this could be an accountability problem given that the sender and message time are incorrect. The valid traces that can be reported in this case are $pt : a_2 \xrightarrow{mid_a} u_1$, $pt : a_3 \xrightarrow{mid_b} u_1$, and $pt : a_2 \xrightarrow{mid_a} u_1 \xrightarrow{mid_c} a_4$.

Notice that the prefix of any valid trace is also a valid trace (though the reporter would be different in each case), but suffixes of a valid trace are *not* always valid (second example). Also the examples highlight the importance of tracing particular messages, not just plaintexts, as we want the platform to be able to reliably associate metadata (senders, receivers, timing) of messages to a reported trace.

We associate to any tracing scheme MT, number of honest users n, and adversary $\mathcal{A}$ the path traceback forging advantage

$$\mathbf{Adv}^{\text{tr-unf}}_{\mathsf{MT},n,\mathcal{A}}(\lambda) = \Pr\left[\text{TrUnf}^{\mathcal{A}}_{\mathsf{MT},n}(\lambda) = 1\right] ,$$

where the probability is taken over the random choices made in the game, including those made by the adversary.

Modifications for tree traceback. The unforgeability game remains largely the same, given in Figure 3.8 highlighted in blue. We add a recursive predicate check on all subtrees returned in the trace. Furthermore, the `WasRec` predicate is set even if a received message was not accepted. This captures the fact that a message will be traced as sent to a user regardless of whether the message

was accepted.

3.2.3 Construction

In this section, we present two constructions for message traceback targeting different traceback goals: path traceback and tree traceback.

Path Traceback

We start with path traceback. The goal is to allow reporting a message with plaintext pt, with the platform then able to identify the sequence of forwarded messages back to the original author of pt. In this case, $\mathsf{Process}$ outputs a trace

$$tr = (tr_1, mid_{1,2}, tr_2, mid_{2,3}, tr_3, \ldots, mid_{\tau-1,\tau}, tr_\tau)$$

where τ is called the trace length and each $tr_i \in \mathcal{U}$ identifies a user and each $mid_{i,j}$ is an identifier for a message. These message identifiers correspond to the ones output by the platform tag processing algorithm ($\mathsf{Process}$), allowing the platform to store, and later recover during traceback, any desired metadata associated with a sent message. This can be visualized as a directed graph where nodes are associated to users and edges to messages. The trace can then be denoted via

$$pt\ :\ tr_1 \overset{mid_{1,2}}{\longrightarrow} tr_2 \overset{mid_{2,3}}{\longrightarrow} \cdots \overset{mid_{\tau-1,\tau}}{\longrightarrow} tr_\tau$$

where pt represents the plaintext traced and the arrow diagram the path.

As discussed in Section 3.2.1, an adversarial user can always obfuscate the source from which they received a message by a partition attack, in which case path traceback will result in identifying the first misbehaving user (from the end) as the originator. For example, if tr_2 behaved maliciously, they can deviate from the proper client implementation and prevent traceback from identifying tr_1, and instead tr_2 would be considered the source of the message.

The linked tags scheme. Each message sent between two users is associated with a *message identifier*, denoted by mid. The message identifier is chosen by the sender, who samples a random *tracing key* k and calculates the message identifier as the output of a PRF on the plaintext, $\mathsf{PRF.Ev}(k, pt)$. In this manner, the message identifier also acts as a commitment to the plaintext, and the tracing key acts as an opening key. Looking forward, our trace unforgeability property will rely on the collision

$\underline{\mathsf{LT.Setup}(\lambda)}$
Return λ

$\underline{\mathsf{LT.InitU}(\lambda)}$
Return $\bot$

$\underline{\mathsf{LT.InitP}(\lambda)}$
Return $T_P \leftarrow [\cdot]$

$\underline{\mathsf{LT.NewMsg}(u, pt : \bot)}$
$k \leftarrow\!\!\$ \{0,1\}^\lambda$
Return k

$\underline{\mathsf{LT.TagGen}(u_s, u_r, pt, k_{i-1} : \bot)}$
$k_i \leftarrow\!\!\$ \{0,1\}^\lambda$
$\widetilde{k}_i \leftarrow \mathsf{H}(k_i)$
$mid \leftarrow \mathsf{PRF.Ev}(k_i, pt)$
$ct \leftarrow \mathsf{SE.Enc}(\widetilde{k}_i, k_{i-1})$
$tt_s \leftarrow (mid, ct)$
Return (k_i, tt_s)

$\underline{\mathsf{LT.RecMsg}(k_i, u_s, u_r, pt, tt_r : \bot)}$
$mid \leftarrow tt_r$
If $mid \neq \mathsf{PRF.Ev}(k_i, pt)$ then return $\bot$
Return k_i

$\underline{\mathsf{LT.Process}(u_s, u_r, tt_s : T_P)}$
$(mid, ct) \leftarrow tt_s$
Require $mid \notin T_P$
$tt_p \leftarrow (ct, u_s, u_r)$
$T_P[mid] \leftarrow tt_p$
$tt_r \leftarrow mid$
Return $((mid, tt_p), tt_r)$

$\underline{\mathsf{LT.Trace}(u, pt, k : T_P)}$
$tr \leftarrow [\cdot]\,;\ i \leftarrow 0$
$tr[i] \leftarrow u\,;\ mid \leftarrow \mathsf{PRF.Ev}(k, pt)$
While $mid \in T_P$ do
  $(ct, u_s, u_r) \leftarrow T_P[mid]$
  If $u_r \neq tr[i]$ then break loop
  $tr[i+1] \leftarrow mid\,;\ tr[i+2] \leftarrow u_s$
  $\widetilde{k} \leftarrow \mathsf{H}(k)\,;\ k \leftarrow \mathsf{SE.Dec}(\widetilde{k}, ct)$
  $mid \leftarrow \mathsf{PRF.Ev}(k, pt)$
  $i \leftarrow i + 2$
Return tr^{-1}

Figure 3.9: Linked tags construction $\mathsf{LT}[\mathsf{PRF}, \mathsf{SE}]$ for path traceback. It is parameterized by a PRF PRF and symmetric encryption scheme SE that we assume has keyspace $\mathsf{K} : \{0,1\}^\lambda$, which we also assume to be the output space of hash function H.

resistance of the PRF to bind message identifiers to a plaintext and tracing key. To link the message as a forward of a previous message, the sender also encrypts the previous message's tracing key with the tracing key for the new message. If the message is not a forward, the sender samples and encrypts a random value. This ciphertext acts as an encrypted pointer to the previous message's identifier. The current message identifier and the encrypted pointer are sent to the platform and are stored in a key-value table in server state, keyed by the message identifier. The platform sends the message identifier to the recipient, who verifies the commitment is well-formed with respect to the tracing key and plaintext before accepting the message.

Traceback is then simply a matter of decrypting and following the pointers between message identifiers in server state. Given a report consisting of a tracing key k_τ and plaintext pt, the platform will lookup $mid_{\tau-1,\tau} = \mathsf{PRF.Ev}(k_\tau, pt)$ in server state and decrypt the encrypted pointer to learn the tracing key $k_{\tau-1}$. Tracing key $k_{\tau-1}$ is in turn used to lookup $mid_{\tau-2,\tau-1} = \mathsf{PRF.Ev}(k_{\tau-1}, pt)$, the previous message in the forwarding chain. The chain ends when a lookup of k_1 fails, i.e., the value $mid = \mathsf{PRF.Ev}(k_1, pt)$ is not found in the server state. Pseudocode for the construction and a diagram of one step of traceback is given in Figure 3.9.

Our scheme can be thought of as a sort of secret share between the platform and the recipient. The recipient gets the tracing key and the plaintext, while the platform gets the ciphertext containing the previous message's tracing key. User trace confidentiality is preserved from the recipient's share

as it has no dependence on the previous message. Platform trace confidentiality is preserved from the platform's share as the message identifier and ciphertext appear as random bytes without knowledge of the tracing key. The two shares combined allow for the previous message's tracing key to be decrypted and traceback to proceed.

The linked tags scheme provides path traceback cheaply. It does require $\mathcal{O}(m)$ storage at the platform for m the total number of messages sent by users. But storage is relatively cheap, and this is a write-heavy workload, potentially allowing cheaper storage options. Of course, the platform can expunge tracing tags after a predefined time (e.g., one week or one month), allowing tracing in the interim but not after. This may be preferable since it improves confidentiality in the long term, but still allows platforms to respond to pressing issues such as an ongoing misinformation campaign targeting candidates within an election.

Tree Traceback

Next, we consider an alternative traceback goal, tree traceback. The goal in tree traceback is to allow reporting of a message with plaintext pt, enabling the platform to identify not just the path of forwarded messages to the original author, as in path traceback, but the entire forwarding tree of messages for pt rooted at the original author. A tree is denoted as a tuple of user identifier and list of children subtrees, where each element of the children subtree list is recursively a tree, i.e., a user identifier and list of children subtrees, along with the message identifier for the message sent between parent and child:

$$tr = \Big(tr_{\mathrm{a}}, \Big[\big(mid_{\mathrm{a},0}, tr_b, [\ldots]\big), \big(mid_{\mathrm{a},1}, tr_c, [\ldots]\big), \ldots\Big]\Big)$$

where $tr_\alpha \in \mathcal{U}$ identifies a user and each $mid_{\alpha,i}$ is an identifier for a message sent by tr_α.

The doubly-linked tags scheme. The doubly-linked tags construction for tree traceback extends the strategy taken in path traceback of storing encrypted pointers between message identifiers. In the path traceback construction, for each message between a sender and recipient, the platform stored an encrypted pointer to trace backwards to the previous message, i.e., where the sender received the message from. Intuitively, in tree traceback, we need to extend this approach to also trace forwards in order to build the forwarding tree. This includes storing pointers to forwards made by the sender *and* forwards made by the recipient. However, attempting to explicitly store encrypted pointers to

other forwards is problematic as the number of forwards of a message are not known at the time of sending; the recipient has not yet even received the message, let alone forwarded it, and the sender could choose to forward the message to new recipients in the future.

We address this challenge by efficiently representing an unbounded set of pointers with a PRF key, gk, which acts as a generator for all the tracing keys associated to forwards of a particular message. One enumerates the tracing keys by evaluating the PRF on a counter ctr which is stored in the client's state, $k \leftarrow \mathsf{PRF.Ev}(gk, ctr)$. As in the path traceback construction, a tracing key points to a message identifier through the evaluation of a PRF, $mid \leftarrow \mathsf{PRF.Ev}(k, pt)$. Thus, the platform stores three encrypted values with each message identifier: (1) an encrypted tracing key $ct_{k_{i-1}}$ for the previous message; (2) an encrypted tracing key generator ct_{gk_i} for other forwards by the sender; and (3) an encrypted tracing key generator $(ct_{gk_{i+1}}, ks_1)$ for forwards by the recipient.

The three encrypted values stored by the platform correspond to three stages of performing tree traceback, each illustrated in Figure 3.10. Given a report consisting of a tracing key and plaintext, first the platform follows the tracing keys, k_{i-1}, for previous message identifiers until reaching the message source, essentially performing path traceback (Figure 3.10 (a)). Next, using the sender tracing key generator, gk_i, the platform enumerates the tracing keys and message identifiers for all sends of the message from the source sender (Figure 3.10(b)). Lastly, for each of these message identifiers, the platform recursively builds out a subtree by enumerating the tracing keys and message identifiers for the recipient's forwards using the recipient tracing key generator, gk_{i+1} (Figure 3.10 (c)). The full pseudocode for the scheme is also given in Figure 3.10.

There are a few subtle design points to our tree traceback scheme that we highlight here. The first concern is how to securely escrow the three encrypted values for each message identifier with the platform. The sender can encrypt and send the previous message's tracing key and its own tracing key generator, but the sender cannot know and therefore cannot escrow the recipient's tracing key generator. For confidentiality, the recipient's tracing key generator should only be known by the recipient. Instead, the sender and the platform each create key shares for the recipient tracing key, ks_0 and ks_1, such that neither the sender nor the platform learn the key, but the sender's key share needed to derive the key is stored encrypted on the platform. The recipient derives their tracing key generator as $gk_{i+1} \leftarrow \mathsf{H}(ks_0 \,\|\, ks_1)$ which appears random (given that the sender and platform are not colluding).

$\underline{\mathsf{DLT.Setup}(\lambda)}$
Return λ

$\underline{\mathsf{DLT.InitU}(\lambda)}$
Return $T_u \leftarrow [\cdot]$

$\underline{\mathsf{DLT.InitP}(\lambda)}$
Return $T_P \leftarrow [\cdot]$

$\underline{\mathsf{DLT.NewMsg}(u, m : T_u)}$
$k_{i-1} \leftarrow\!\!\$\, \{0,1\}^\lambda$
$gk_i \leftarrow\!\!\$\, \{0,1\}^\lambda$
$tmd \leftarrow (k_{i-1}, gk_i)$
$T_u[tmd] \leftarrow 0$
Return tmd

$\underline{\mathsf{DLT.TagGen}(u_s, u_r, pt, tmd : T_u)}$
Require $tmd \in T_u$
$ctr \leftarrow T_u[tmd]$; $T_u[tmd] \leftarrow ctr + 1$
$(k_{i-1}, gk_i) \leftarrow tmd$
$k_i \leftarrow \mathsf{PRF.Ev}(gk_i, ctr)$
$mid \leftarrow \mathsf{PRF.Ev}(k_i, pt)$
$ks_0 \leftarrow\!\!\$\, \{0,1\}^\lambda$; $\widetilde{k}_i \leftarrow \mathsf{H}(k_i)$
$ct_{k_{i-1}} \leftarrow \mathsf{SE.Enc}(\widetilde{k}_i, k_{i-1})$
$ct_{gk_i} \leftarrow \mathsf{SE.Enc}(\widetilde{k}_i, gk_i)$
$ct_{gk_{i+1}} \leftarrow \mathsf{SE.Enc}(\widetilde{k}_i, ks_0)$
$tt_s \leftarrow (mid, ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}})$
Return (T_u, k_i, tt_s)

$\underline{\mathsf{DLT.RecMsg}(k_i, u_s, u_r, pt, tt_r : T_u)}$
$(mid, ct_{gk_{i+1}}, ks_1) \leftarrow tt_r$
If $mid \neq \mathsf{PRF.Ev}(k_i, pt)$ then return $\perp$
$\widetilde{k}_i \leftarrow \mathsf{H}(k_i)$
$ks_0 \leftarrow \mathsf{SE.Dec}(\widetilde{k}_i, ct_{gk_{i+1}})$
$gk_{i+1} \leftarrow \mathsf{H}(ks_0 \,\|\, ks_1)$
$tmd \leftarrow (k_i, gk_{i+1})$
$T_u[tmd] \leftarrow 0$
Return tmd

$\underline{\mathsf{DLT.Process}(u_s, u_r, tt_s : T_P)}$
$(mid, ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}) \leftarrow tt_s$
Require $mid \notin T_P$
$ks_1 \leftarrow\!\!\$\, \{0,1\}^\lambda$
$tt_p \leftarrow (ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, u_s, u_r)$
$tt_r \leftarrow (mid, ct_{gk_{i+1}}, ks_1)$
Return $((mid, tt_p), tt_r)$

$\underline{\mathsf{DLT.Trace}(u, pt, tmd : T_P)}$
$(k_i, gk_{i+1}) \leftarrow tmd$
$root \leftarrow u$
$root_{gk} \leftarrow gk_{i+1}$
$mid \leftarrow \mathsf{PRF.Ev}(k_i, pt)$
While $mid \in T_P$ do
    $(ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, u_s, u_r) \leftarrow T_P[mid]$
    If $root \neq u_r$ then break loop
    $\widetilde{k}_i \leftarrow \mathsf{H}(k_i)$
    $ks_0 \leftarrow \mathsf{SE.Dec}(\widetilde{k}_i, ct_{gk_{i+1}})$
    If $\neg\mathsf{valid}_{gk}(gk_{i+1}, ks_0, ks_1)$ then break loop
    $gk_i \leftarrow \mathsf{SE.Dec}(\widetilde{k}_i, ct_{gk_i})$
    If $\neg\mathsf{valid}_k(k_i, gk_i, T_P)$ then
        Return $(u_s, [(\mathsf{PRF.Ev}(k_i, pt), \mathsf{trace_fwd}(T_P, pt, u_r, gk_{i+1}))])$
    $root \leftarrow u_s$
    $root_{gk} \leftarrow gk_i$
    $k_i \leftarrow \mathsf{SE.Dec}(\widetilde{k}_i, ct_{k_i})$; $gk_{i+1} \leftarrow gk_i$
    $mid \leftarrow \mathsf{PRF.Ev}(k_i, pt)$
Return $\mathsf{trace_fwd}(pt, root, root_{gk} : T_P)$

$\underline{\mathsf{trace_fwd}(pt, u, gk_i : T_P)}$
$tr \leftarrow [\cdot]$; $j \leftarrow 0$
While $k \leftarrow \mathsf{PRF.Ev}(gk_i, j); mid \leftarrow \mathsf{PRF.Ev}(k, pt);\ mid \in T_P$ do
    $(ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, u_s, u_r) \leftarrow T_P[mid]$
    $\widetilde{k} \leftarrow \mathsf{H}(k)$
    $ks_0 \leftarrow \mathsf{SE.Dec}(\widetilde{k}, ct_{gk_{i+1}})$
    $gk_{i+1} \leftarrow \mathsf{H}(ks_0 \,\|\, ks_1)$
    $tr[j] \leftarrow (mid, \mathsf{trace_fwd}(T_P, pt, u_r, gk_{i+1}))$
    $j \leftarrow j + 1$
Return (u, tr)

Figure 3.10: Doubly-linked tags construction $\mathsf{DLT}[\mathsf{PRF}, \mathsf{SE}]$ for tree traceback. It is parameterized by a PRF PRF and symmetric encryption scheme SE that we assume has keyspace $\mathsf{K} : \{0,1\}^\lambda$, which we also assume to be the output space of hash function H.

A second concern arises from the enumeration of tracing keys from a generator during traceback. In stages (b) and (c) in Figure 3.10, the platform enumerates tracing keys by evaluating a PRF keyed by the escrowed generator, gk, on a counter initialized to zero, incrementing and re-evaluating to produce the next tracing key. This continues until the produced tracing key does not evaluate to a valid message identifier in platform storage, indicating all forwarding branches for the user have been enumerated. This traceback approach will only succeed if users correctly derive tracing key generators from the escrowed key shares and correctly derive tracing keys from the generator by

incrementing a counter, and not, for example, skipping a counter value. Left as is, these types of deviations would result in a class of partition attacks that are so-called "unidirectional". In these attacks, a malicious user is able to partition the tree trace to hide a subtree such that a report in the main tree will end at the malicious user and not include the subtree; but at the same time, a report in the subtree will trace through the malicious user and identify the main tree as the source.

We address this by enforcing that a message is only traced back to a sender if it would have also been traced forward to the recipient. This invariant restricts malicious users to only being able to mount "complete" partition attacks, in which if they choose to partition, they are implicated in two disjoint traces: the end of one trace and the source of the other. Enforcing this invariant manifests in tree traceback by two well-formedness checks (denoted valid_{gk} and valid_{k} in the pseudocode). The first check simply rederives the recipient generator to make sure it matches the one escrowed by the recipient. The second well-formedness check determines if a tracing key was properly derived from a generator. Doing so requires enumerating with a counter, succeeding when the current message tracing key is found, or fails when a generated tracing key evaluates to an invalid message identifier. Both of these checks take place during stage (a) of tree traceback to identify a root that will not include fragmented subtrees.

3.2.4 Security Analysis

We anlayze the security of our path traceback and tree traceback schemes with respect to our proposed security definitions.

Platform trace confidentiality. The sender trace tag in our path traceback scheme is made up of a message identifier, which is the output of a PRF, and a ciphertext. Intuitively, since the platform does not learn the key used with the PRF or with encryption scheme, our scheme satisfies the security property. More formally,

Theorem 3. *Let* LT *be the message tracing scheme for path traceback defined in Figure 3.9 using hash function* H*. Then if* H *is modeled as a random oracle, for any* $\textsc{PTrConf}$ *adversary* $\mathcal{A}_{\text{p-tr-conf}}$ *that makes at most* q *queries to the challenge oracle, we give adversary* $\mathcal{A}_{\text{prf}}$ *and* $\mathcal{A}_{\text{rorcpa}}$ *such that*

$$\mathbf{Adv}^{\text{p-tr-conf}}_{\mathsf{LT},\mathcal{A}_{\text{p-tr-conf}}}(\lambda) \le \mathbf{Adv}^{\text{prf}}_{\mathsf{PRF},q,\mathcal{A}_{\text{prf}}}(\lambda) + \mathbf{Adv}^{\text{rorcpa}}_{\mathsf{SE},q,\mathcal{A}_{\text{rorcpa}}}(\lambda)$$

where if $\mathcal{A}_{\text{p-tr-conf}}$ runs in time T, then $\mathcal{A}_{\text{prf}}$ and $\mathcal{A}_{\text{rorcpa}}$ run in time $T' \approx T$ and $\mathcal{A}_{\text{prf}}$ makes at most q oracle queries.

Proof sketch: The proof proceeds in a straightforward fashion through two main game hops. The first replaces the PRF evaluation with that of a random function and bounds the distinguishing advantage by the PRF security of PRF. The second replaces the encryption output with a random bit string and bounds the distinguishing advantage by the real-or-random security of the underlying encryption scheme. After these two steps, it is easy to see that the sender trace tag, which consists of the output mid of PRF and a ciphertext ct, is a random bit string.

A similar approach is taken for the tree traceback scheme, formalized as a theorem statement below.

Theorem 4. *Let DLT be the message tracing scheme for tree traceback defined in Figure 3.10 using hash function H. Then if H is modeled as a random oracle, for any PTrConf adversary $\mathcal{A}_{\text{p-tr-conf}}$ that makes at most q_{new} queries to the message oracles and q challenge queries, we give adversary $\mathcal{A}_{\text{prf}}$ and $\mathcal{A}_{\text{rorcpa}}$ such that*

$$\mathbf{Adv}^{\text{p-tr-conf}}_{\mathsf{DLT},\mathcal{A}_{\text{p-tr-conf}}}(\lambda) \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{PRF},q_{\mathsf{new}}+q,\mathcal{A}_{\text{prf}}}(\lambda) + \mathbf{Adv}^{\text{rorcpa}}_{\mathsf{SE},q,\mathcal{A}_{\text{rorcpa}}}(\lambda) + \frac{q_{\mathsf{new}}^2}{2^n}$$

where if $\mathcal{A}_{\text{p-tr-conf}}$ runs in time T, then $\mathcal{A}_{\text{prf}}$ and $\mathcal{A}_{\text{rorcpa}}$ run in time $T' \approx T$ and $\mathcal{A}_{\text{prf}}$ make at most $3q$ oracle queries.

Proof sketch: The proof proceeds with the same strategy as for path traceback. The first transition replaces the PRF executions with executions of a random function, where we can bound the distinguishing advantage by the PRF security of PRF since the PRF keys are random and not revealed to the adversary. There is a subtlety here that since the generator can be reused across multiple challenge oracle calls (with an incremented counter), it can be distinguished from random if a generator is every resampled. Thus, we include a term bounding the low probability event of a generator resampling collision. The second transition replaces calls to the encryption algorithm with sampling random bits. After these transitions, the sender trace tag output from the challenge oracle is a mid which is the output of a random function, and three ciphertexts which are random bit strings.

The full proofs are deferred to the full version [TMR19b].

User trace confidentiality. In our path traceback scheme, the recipient's view consists of the message

identifier mid, tracing key k_i, and the plaintext. Importantly, the ciphertext ct is stored by the platform and not visible to the recipient. The tracing key is randomly generated for each sent message and the message identifier is calculated as a function of the tracing key and plaintext, $mid \leftarrow \mathsf{PRF.Ev}(k_i, pt)$. Thus, no part of the recipient's view is dependent on previous message trace metadata, e.g., k_{i-1}, and any adversary's advantage against our scheme is 0.

Theorem 5. *Let* LT *be the message tracing scheme for path traceback defined in Figure 3.9. For any* $\mathrm{UTRCONF}$ *adversary* $\mathcal{A}$*,*

$$\mathbf{Adv}^{\text{u-tr-conf}}_{\mathsf{LT},\mathcal{A}}(\lambda) = 0 \,.$$

In tree traceback, the sender trace tag and tracing key are similarly independent of the challenge bit leading again to an advantage of zero.

Theorem 6. *Let* DLT *be the message tracing scheme for path traceback defined in Figure 3.10. For any* $\mathrm{UTRCONF}$ *adversary* $\mathcal{A}$*,*

$$\mathbf{Adv}^{\text{u-tr-conf}}_{\mathsf{DLT},\mathcal{A}}(\lambda) = 0 \,.$$

Trace unforgeability. Intuitively, trace unforgeability is achieved in our path traceback scheme due to the binding of the plaintext to a message identifier with the tracing key. Honest users check the binding of message identifiers they receive and send, so an adversary that wishes to frame a user must find a collision on one of the honest user's message identifiers. For example, to achieve the message replacement attack described above, an adversary must find an alternate plaintext and key that collides with the honest user's sent message identifier. We thus provide the following theorem statement using the collision resistance and PRF security of PRF. We provide a proof sketch here and defer the complete proof to the full version [TMR19b].

Theorem 7. *Let* LT *be the message tracing scheme for path traceback defined in Figure 3.9. Then, for any* TRUNF *adversary* $\mathcal{A}_{\text{tr-unf}}$ *that makes at most* q_{new} *new message queries and* q_{send} *queries, to the send oracles, we give adversaries* $\mathcal{A}_{\text{prf}}$ *and* $\mathcal{A}_{\text{cr}}$ *such that*

$$\mathbf{Adv}^{\text{tr-unf}}_{\mathsf{LT},n,\mathcal{A}_{\text{tr-unf}}}(\lambda) \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{PRF},q_{\mathsf{new}},\mathcal{A}_{\text{prf}}}(\lambda) + \mathbf{Adv}^{\text{cr}}_{\mathsf{PRF},\mathcal{A}_{\text{cr}}}(\lambda) + \frac{q_{\mathsf{send}}}{2^n}$$

where if $\mathcal{A}_{\text{tr-unf}}$ runs in time T, then $\mathcal{A}_{\text{prf}}$ and $\mathcal{A}_{\text{cr}}$ run in time $T' \approx T$ and $\mathcal{A}_{\text{prf}}$ makes at most q_{send} oracle queries.

Proof sketch: This proof proceeds as a careful case analysis of the four adversary winning conditions. We show that nearly all of the winning conditions correspond to an adversary finding a collision in PRF. In the single subcase that does not result in a collision, we argue that the probability of reaching this subcase corresponds to guessing the output of PRF keyed by a tracing key sampled in NEWMSG. Since tracing keys sampled in NEWMSG are never revealed to the adversary, we can show this probability is low using the PRF security of PRF.

Theorem 8. *Let DLT be the message tracing scheme for tree traceback defined in Figure 3.10. Then, for any TRUNF adversary $\mathcal{A}_{\text{tr-unf}}$ that makes at most q_{new} new message queries and q_{send} queries, to the send oracles, we give adversary $\mathcal{A}_{\text{cr}}$ such that*

$$\mathbf{Adv}^{\text{tr-unf}}_{\mathsf{DLT},n,\mathcal{A}_{\text{tr-unf}}}(\lambda) \leq \mathbf{Adv}^{\text{cr}}_{\mathsf{PRF},\mathcal{A}_{\text{cr}}}(\lambda) + \frac{(q_{\mathsf{new}} + q_{\mathsf{send}})^2}{2^n}$$

where if $\mathcal{A}_{\text{tr-unf}}$ runs in time T, then $\mathcal{A}_{\text{cr}}$ runs in time $T' \approx T$.

Proof sketch: The proof strategy is the same as in trace unforgeability for path traceback. We will perform a case analysis of the three winning conditions of the security game and show that in each one, we will either be able to show a collision of PRF or can show that it is a low probability event by the PRF security of PRF.

- Case 1: honest root did not author message

  $$\texttt{WasSent}(tr_0, tr_1, mid_{0,1}, pt^*) = \mathsf{false}$$

  For $mid_{0,1}$ to be added to the trace, we know $T_P[mid_{0,1}]$ is populated and has tr_0 as the sender. Given the oracles in our game, the only way to populate T_P with $tr_0 \in [1, n]$ as sender is through the honest send oracle SENDMAL which was called with a message id counter s indicating sending plaintext pt_s. There are two subcases: (Case 2a) $pt_s \neq pt^*$, or (Case 2b) $pt_s = pt^*$.

  - Case 1a: $pt_s \neq pt^*$

    We consider the tracing key generated during tag generation in the oracle call; call this $k_{0,1}$. We will also consider the tracing key used in traceback to include $mid_{0,1}$ in the trace, call

this $trk_{0,1}$. We know from the execution of SEND that $\mathsf{PRF.Ev}(k_{0,1}, pt_s) = mid_{0,1}$. We also know from the execution of Trace that for $mid_{0,1}$ to have been included in the trace, there must have been a tracing key $trk_{0,1}$ used such that $\mathsf{PRF.Ev}(trk_{0,1}, pt^*) = mid_{0,1}$. This constitutes a collision.

- Case 1b: $pt_s = pt^*$

  For tr_0 to have been identified as the root of the trace, there must have been some mid_b that had tr_0 as the sender. The root identifying condition is either that the escrowed tracing key evaluates to an invalid mid or that the current tracing key is not well-formed with respect to the escrowed generator. Since tr_0 is honest, we can rule out the second case, and the first case would only occur if tr_0 was the author of the message — since our oracles do not make it possible for an honest user to forward a message that was not received from the server and accepted by RecMsg. So tr_0 is the author of a s' counter associated with pt^*. Any call to SEND of s' would result in the `WasSent` predicate being sent to true since s' was authored. Thus, $s \neq s'$. This means that either the generator for s and the generator for s' collide on a tracing key, the tracing key from s and the tracing key from s' collide on $mid_{0,1}$, or the generators are the same for s and s'. The last case we can bound as a low probability event.

- Case 2: honest leaf did not receive message

$$\mathtt{WasRec}(tr_{i-1}, tr_i, mid_{i-1,i}, pt^*) = \mathsf{false}$$

To be added to the trace, it must be that $mid_{i-1,i}$ in T_P contained tr_{i-1} as sender and tr_i as the recipient. From our oracle construction, the only way for a pair of users to be added to T_P is through the SEND or SENDMAL oracles, which both set the `WasRec` predicate with plaintext pt input to the oracle. In which case, if $pt^* = pt$, the `WasRec` would have been true. Therefore, $pt^* \neq pt$ and there exists a collision with the tracing key used in traceback trk,pt^* and the tracing key used in the oracle k,pt on $mid_{i-1,i}$.

- Case 3: internal node did not forward message

$$\mathtt{WasFwd}(tr_i, tr_{i+1}, mid_{i-1,i}, mid_{i,i+1}, pt^*) = \mathsf{false}$$

Same as in Case 1, for $mid_{i,i+1}$ to be added to the trace, we know $T_P[mid_{i,i+1}]$ is populated

and has tr_i as the sender. Given the oracles in our game, the only way to populate T_P with $tr_i \in [1, n]$ as sender is through the honest send oracle $\textsc{SendMal}$ which was called with a message id counter s indicating sending plaintext pt_s. There are two subcases: (Case 2a) $pt_s \neq pt^*$, or (Case 2b) $pt_s = pt^*$.

- Case 3a: $pt_s \neq pt^*$

  We consider the tracing key generated during tag generation in the oracle call; call this $k_{i,i+1}$. We will also consider the tracing key used in traceback to include $mid_{0,1}$ in the trace, call this $trk_{i,i+1}$. We know from the execution of $\textsc{Send}$ that $\mathsf{PRF.Ev}(k_{i,i+1}, pt_s) = mid_{i,i+1}$. We also know from the execution of Trace that for $mid_{i,i+1}$ to have been included in the trace, there must have been a tracing key $trk_{i,i+1}$ used such that $\mathsf{PRF.Ev}(trk_{i,i+1}, pt^*) = mid_{i,i+1}$. This constitutes a collision.

- Case 3b: $pt_s = pt^*$

  We will further consider two subsubcases. If $mid_{i-1,i} = mid_s$, then since the `WasFwd` was not set to true, we can infer that s was authored by tr_i and importantly, gk was sampled randomly. If $mid_{i-1,i} \neq mid_s$, then we can infer gk was the hashed output of the key shares provided for mid_s, which importantly, likely differ from the key shares provided by $mid_{i-1,i}$ created gk'. This means that there is a collision between gk,ctr and gk',ctr' to $k_{i,i+1}$, a collision between $k_{i,i+1}$,pt^* and $trk_{i,i+1}$,pt^* to $mid_{i,i+1}$, or gk and gk' are the same which we can bound to be low probability.

3.2.5 Evaluation

Implementation. To evaluate our tracing protocols, we provide a prototype library and tracing service implementation in Rust that can be readily integrated into existing end-to-end encrypted messaging systems. For our hash, collision-resistant pseudorandom function, and block cipher primitives, we use SHA-3, HMAC derived from SHA-3, and AES-128. All of these operations are supported by the Rust Crypto library. The tracing service uses Redis as its underlying key-value store. Both the library and service code are available open source at `https://github.com/nirvantyagi/tracing`.

Our tracing service can be integrated into existing end-to-end encrypted messaging systems with

the following client and server side changes. The client will make library calls to NewMsg, TagGen, and RecMsg when sending, forwarding, and receiving messages and it will store trace metadata associated with messages accepted by RecMsg. The tracing key is included with the plaintext in the end-to-end encrypted ciphertext. The sender and recipient trace tags are sent alongside the end-to-end ciphertext to and from the platform. On the server side, the tracing service is run as an internal service. The messaging server receives the end-to-end ciphertext and sender trace tag and sends a "process" request including the sender trace tag and user identifiers for the sender and recipient to the tracing service. The tracing service handles running Process, storing the appropriate information in a key-value store, and returns the recipient trace tag, which the messaging server delivers with the end-to-end ciphertext. The client and messaging server also need to be modified to send and accept reports of messages (if that functionality is not already included). The client includes the trace metadata along with the plaintext in the report. The messaging server simply forwards the reported plaintext and trace metadata to the tracing service which runs Trace and saves the message trace to be used downstream for moderation.

Timing benchmarks. At a high level, our tracing schemes are fast and induce minimal storage and bandwidth overhead. This is to be expected as our schemes are composed of symmetric cryptographic techniques over small 128 bit components. Experiments were performed on a 2.2 GHz Intel Core i7 Processor with 8 GB of RAM. The time to run the client-side algorithms, TagGen and RecMsg, for generating and verifying trace tags is shown in Figure 3.11; the NewMsg algorithm is not shown as it simply samples a random number. For path traceback the tag generation and verification algorithms take < 10 microseconds, and for tree traceback the algorithms take < 50 microseconds. In practice, client side operations will often be performed on less powerful mobile devices, e.g. running ARM processors, but we do not expect the difference in timing to be prohibitive.

We next turn to evaluate the server side algorithms, Process and Trace. The server side algorithms interact with a key-value store, the performance characteristics of which will affect the performance of the algorithm. In our schemes, the Process algorithm essentially performs a key-value put operation and relays the trace tag; no cryptographic operations are performed. As expected, this translates to a minimal cost operation for most key-value stores – in our benchmarks, in which the server key-value store is instantiated with an in-memory Redis data store, the Process algorithm takes on the order

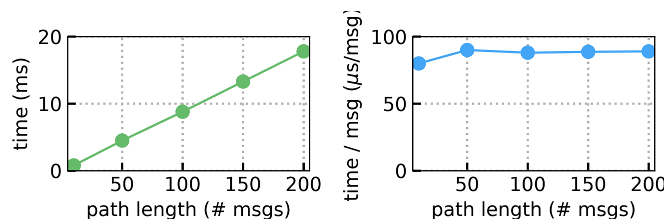

Figure 3.11: Path traceback timing with respect to path length. (Left) Total time to complete trace. (Right) Traceback rate of time per message in trace.

of 100 microseconds. Building a trace of messages using the Trace algorithm is where the majority of computation is performed. Importantly, the time to build a trace is linearly dependent on the number of messages revealed in the trace, i.e. trace size. Intuitively, this is because our schemes traverse the trace message by message performing only a constant number of decryptions and PRF evaluations per message. This relationship is easily seen for the case of path traceback as shown in Figure 3.11, in which we find that building a trace takes ≈ 100 microseconds per message in the path. For tree traceback, the traceback time is dependent not only on tree size, but also on tree structure; in particular, the branching factor, i.e., the average number of forwards made by each user. In our tree traceback scheme, the branching factor is the number of forward tracing keys learned per message lookup and decryption. The cost of PRF evaluations to enumerate forward tracing keys is less expensive than the key-value lookup and decryption cost of dereferencing a tracing key. Thus, as the branching factor increases, the time per message decreases (Figure 3.12 (right)). In the worst case, with branching factor equal to one, i.e. a path, the time per message is < 300 microseconds, which still leads to efficient tree traces regardless of structure. As an example, a trace of a tree of size 20,000 is built in under two seconds.

Storage and bandwidth overhead. Our tracing schemes introduce extra tracing metadata that needs to be stored and send by both the client and server. The absolute size of the stored trace metadata is small — a 256-bit PRF output and a few 128-bit block cipher outputs. For client storage and bandwidth, we expect the overhead induced by < 100B of trace metadata per message to be dwarfed by the size of the message itself; furthermore, for client storage, when the message is deleted, the associated trace metadata can be deleted with it. For server storage, however, in platforms like Signal and WhatsApp, message ciphertexts are not stored, aside from a temporary staging period until they

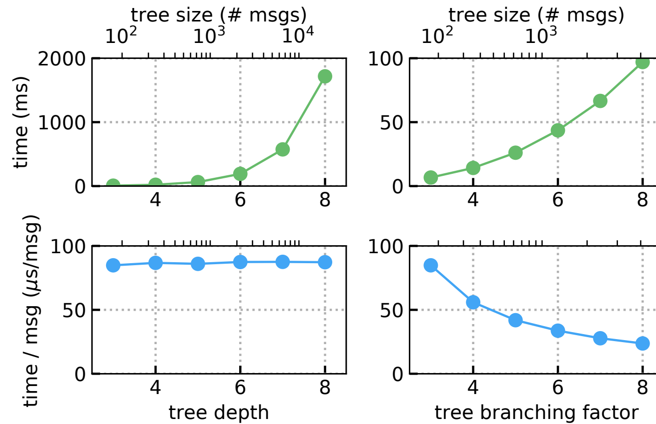

Figure 3.12: Tree traceback with varying tree structure. (Left) Varying tree depth with constant branching factor of 3. (Right) Varying branching factor with constant depth of 3. (Top) Total time to complete trace. (Bottom) Traceback rate of time per message in trace.

have been delivered. In this case, trace metadata incurs the addition of a new long-term storage cost that potentially represents a significant infrastructure change. To limit storage costs, if the goal of message tracing is to combat ongoing misinformation campaigns, it seems reasonable to store only a sliding window of trace metadata, say for the current month. In this case, if the platform sees one billion messages of traffic per day, the data store would be of size $\approx$ 2TB for tree traceback and 600GB for path traceback. A data store of this size can be instantiated with an in-memory data store like Redis as in our benchmarks, or more cheaply with a database, where the tradeoff would be slower traceback.

3.3 Follow-up Work

Since the initial publication of the work presented in this chapter [TGL+19a, TMR19a], a number of follow-up work has been conducted building and improving on the results. One direction is to extend support for abuse reporting to groups of users [LZH+23] and committees of moderators [PH23]. Another set of works build on the problem of message traceback. Source traceback ensures that only the originating source of the forwarded message is revealed on report [PEB21]. FACTS is a system for enabling traceback only given a high threshold of reports of a message [LRTY22]. Hecate [IAV22] and Cerberus [PH23] support the combination of goals of metadata-privacy in AMFs and message traceback. Lastly, another line of work takes a different strategy from user-driven abuse

reporting and considers the problem of automated platform detection of abuse (like misinformation) and the associated transparency mechanisms for safe deployment [KM21, HNC+22, BGJP23, SKM23, TMS+23].

CHAPTER 4

SENDER-ANONYMOUS BLOCKLISTING

End-to-end (E2E) encrypted messaging provides strong E2E confidentiality and integrity guarantees [ACD19, CCD+17]: the messaging platform itself cannot read or modify user messages. The E2E encryption protocols used [PM16] do not, however, attempt to ensure anonymity, so the platform learns the sender and recipient of every message sent over the network. While academic systems [AKTZ17, LYK+19, vdHLZZ15, CBM15, AS16, TGL+17, LGZ18, KLD20, CF10, WCFJ12] have developed protocols that hide the identity of senders and receivers from platforms, they introduce expensive overheads.

A recent suggestion for pragmatic privacy improvements is to aim solely for sender anonymity. Introduced by Signal in a feature called "sealed sender" [Lun17], sender anonymity ensures that the sender's identity is never revealed via messages to the platform, e.g., the sender does not authenticate with an account password or digital signature; messages reveal only the intended recipient. While sealed sender does not hide network-level identifiers such as IP addresses, one can do so by composing it with Tor [DMS04] or an anonymous broadcast [CBM15, WCFJ12, PHE+17, KCDF17, LYK+19].

In this work, we explore a key tension in sender-anonymous systems: mitigating abuse by malicious senders. Already E2E encryption makes some kinds of abuse mitigations, such as content-based moderation, more challenging (c.f., [DGRW18, GLR17, TGL+19a, Fac17]). Sender anonymity complicates the setting further because the lack of sender authentication means that the platform cannot block unwanted messages on behalf of a recipient in a conventional way.

To enable platform blocking, Signal's sealed sender has a user distribute an access key to their contacts that senders must show to the platform when sending the user a sender-anonymous message. If a sender cannot provide an access key, the platform drops the message. A user that blocks a sender in their client triggers a rotation of this key and a redistribution to the (remaining) contacts. Future messages from the blocked sender will be dropped by the platform.

We observe two deficiencies with this approach. First, access keys must be distributed over non-sender-anonymous channels, meaning the platform learns the identities of users who can send sender-anonymous messages to a particular recipient. This significantly lowers the anonymity guarantee—in the limit of having only a single contact, there is no anonymity at all.

Second, we show a simple "griefing" attack that works despite the anti-abuse mechanism. By design, the sender is hidden from the platform, and only the recipient can identify the sender of a sender-anonymous message. However, a malicious sender can trivially craft malformed messages that even the recipient will not be able to identify. The recipient's client rejects these messages, but not before processing them. This is particularly problematic for mobile clients as it uses up battery life; we experimentally verify that an attacker can easily drain a target's battery in a short period of time. To make matters worse, neither victim nor platform can identify the attacker, and so the victim will not know who to block.

We design a new abuse mitigation mechanism for privacy-preserving blocklisting in sender-anonymous messaging. Our protocol, called Orca, allows recipients to register a blocklist with the platform. The blocklist is privacy-preserving, meaning it does not reveal the identities of the blocked users. Senders construct messages that are anonymous to the platform, but can be verified by the platform as being attributable to a sender not present on the blocklist. If the sender is on the blocklist or if the message is malformed, then the platform rejects the message; if the message is delivered, the recipient is guaranteed to be able to identify the sender.

Importantly, Orca provides a new non-interactive initialization functionality that allows a user to initiate sender-anonymous messages without having previously communicated with the recipient. This significantly enhances the anonymity guarantees, because it expands the anonymity set to be as large as all registered users of the system.

In summary, our contributions are:

- We build a threat model for sender-anonymous messaging and identify limitations in previous approaches, including a new griefing attack against Signal's sealed sender that we evaluate.
- We construct a new group signature scheme [CvH91] to make up the core of Orca's functionality. The new primitive is tailored to the needs of our setting and supports multiple openers, keyed verification, and local revocation. We provide new security definitions, building upon ones from prior work [BSZ05, BS04].
- We show an extension of Orca that integrates mechanisms from anonymous credentials [CMZ14] to arrange that the relatively expensive group signature scheme is only used periodically when initiating a new conversation. Initialization will generate a batch of one-time-use sender to-

kens [Lan16, LP16], which can be spent to authenticate messages and replenished at very low cost.

- We implement and evaluate Orca, suggesting that it is sufficiently performant to deploy at scale. In particular, once initialized, the token-based extension incurs only 30B additional bandwidth cost per message and only one extra group exponentiation of computation for clients; the platform need only compute a group exponentiation and check the token against a strikelist. The computational cost for the platform is paid during initialization which incurs work on the order of the size of the recipient's blocklist ($\sim$ 200ms for a blocklist of length 100). We find that a medium-provisioned server can comfortably support a deployment of a million users depending on frequency of conversation initialization.

Related work.

Anonymous credentials. Anonymous credentials [CL04] allow a user to present a cryptographic token proving some specific statement about their identity (e.g., their authorization to send messages to a particular recipient), without revealing anything else about their identity. A problem with anonymous credentials in our setting is that they are — by design — not attributable. While the server processing messages can verify the sender is authorized, the recipient cannot identify the sender. This means there is no way for the server to block the sender in the future, even if some revocation mechanism for the credentials did exist.

A notable design contrast to general-purpose anonymous credential schemes is Privacy Pass [DGS+18], which offers single use credentials that encode only one bit — "I am authorized." Privacy Pass mints tokens using a verifiable oblivious pseudorandom function [JKK14, JKKX16], which is more efficient than our approach of blind MACs [CMZ14], but does not provide the algebraic structure needed to prove relations on the input. We need this property to encrypt the input to the recipient to allow linking of tokens. Blind MACs have been previously suggested for use as one-time tokens [LdV17] and have also been recently proposed as part of Signal's new proposal for private group messaging [CPZ20].

Anonymous blocklisting. Anonymous blocklisting [HG11, TAKS07, TKCS11] systems cover a variety of cryptographic techniques. In general, these systems allow a user to authenticate anonymously to

third parties in such a way that the third party can block them from subsequent authentications if they misbehave. In some systems, this blocking ability takes the form of an additional trusted third party that can de-anonymize users much like a group signature. In others, every time a user authenticates they provide a fresh anonymous cryptographic token derived from their identity and a proof that the current blocklist contains no tokens generated by their own keys. Such systems are cryptographically expensive, requiring work linear in the blocklist for the sender (with the exception of recent work using SNARKs to allow the sender to reuse previous work efficiently [RMM22]). Moreover, much of the overhead across both settings comes from providing anonymity from the third party. Our setting differs in that the sender need not be anonymous (and in fact, should be identifiable) to the party *adding* to the blocklist (i.e., the recipient), but only be anonymous to the party *filtering* on the blocklist (i.e., the platform).

Abuse reporting in E2EE messaging. In relation to the abuse reporting work discussed in the previous chapter which allow the recipient to verifiably reveal the content of a message to the platform to enable content moderation. They allow attribution of message content to a sender for a known sender identity. They do not allow the attribution of a malformed message with unknown sender as in the griefing attack we describe.

Metadata-private messaging. A number of messaging systems have been proposed that provide strong metadata-privacy even against strong network adversaries [AKTZ17, LYK+19, vdHLZZ15, CBM15, AS16, TGL+17, LGZ18, KLD20, CF10, WCFJ12, PHE+17]. These systems incur significant costs on their users, e.g. to send and receive messages at frequent intervals. These costs may dwarf the costs of the types of abuse that Orca aims to prevent. Despite this, a subclass of these systems that could still make use of Orca for blocklisting are based on anonymous broadcasting [CBM15, WCFJ12, PHE+17, KCDF17, LYK+19]. Anonymous broadcasts can be converted to a sender-anonymous messaging service by having a messaging service collect, filter, and deliver the broadcast messages with designated recipients.

4.1 Sender Anonymity in Messaging

This chapter focuses on sender-anonymous E2E encrypted messaging hosted by a centralized messaging platform. In this section and throughout the body, we will often use Signal as our running example.

However, the techniques that we introduce are relevant for any sender-anonymous messaging system in which the platform learns the recipient identity.

4.1.1 Background: Signal and Sealed Sender

Non-sender-anonymous E2EE messaging. We first briefly outline Signal's non-sender-anonymous protocol. For simplicity we restrict attention to one client per user. A user wishing to send a message first registers an account with the platform using a long-lived identity public key pk_s, retaining the associated secret key sk_s. The user then must contact the platform to obtain the long-lived public key pk_r of their intended recipient. Once this phase is complete, a client can securely send messages via Signal's *double ratchet* protocol [PM16]. This provides state-of-the-art message confidentiality guarantees even in the event of key compromise [ACD19, CCD+17].

Signal, like most other E2E encrypted messaging platforms, requires users to authenticate their account when sending and receiving messages. Importantly, this allows for abuse prevention because the platform can block malicious senders, and even block senders from talking to a specific recipient. On the other hand, such account authentication, e.g., via public key signature or unique account password, does not provide cryptographic sender anonymity.

Sender anonymity with sealed sender. Sealed sender is Signal's protocol [Lun17] for cryptographic sender anonymity motivated by their desire to minimize the amount of trust their users must place in the platform. We will now walk through a high level summary of how sealed sender works.

Initialization and key exchange. As before, senders must first register a public key pk_s with the platform. The user is issued a short-lived *sender certificate* from the platform, that we denote by $cert$. The certificate contains a digital signature by the platform in order to attest to the validity of the user's identity key. These certificates must be periodically updated, requiring the user to rerun the registration protocol.

To receive sealed messages a recipient must generate their long-lived identity key pair (pk_r, sk_r) as usual, but now additionally generate a 96-bit *access key* that we denote by ak. Both pk_r and ak are registered with the platform. Looking ahead, senders will need to show ak to the platform to send a sealed message. This means that the recipient must distribute ak to whomever they want to grant

the ability to send sealed messages. By default, the access key is distributed to all contacts of a user through Signal's original non-sender-anonymous channel. Additionally, users can opt into accepting sealed messages from anyone, including non-contacts. In this case, senders do not need a recipient's access key to send them sealed messages.

Sending a sealed message. The pseudocode for sending and receiving a message via sealed sender is provided in Figure 4.1. It is designed to work modularly as a layer on top of any non-sender-anonymous E2E encryption protocol. At a high level, the protocol creates two ciphertexts: (1) an identity ciphertext encrypting the sender's long-lived public key pk_s to the recipient, and (2) a content ciphertext encrypting the standard E2E encryption ciphertext along with the sender certificate. The identity ciphertext and content ciphertext cryptographically hide the sender identity even if the underlying E2E encryption ciphertext does not [1].

More specifically, the protocol encrypts the sender identity pk_s via a variant of hashed ElGamal [ABR01] to produce the identity ciphertext ct_{id}. In particular, it generates ephemeral key pair (pk_e, sk_e) and makes use of a hash-based key derivation function HKDF and authenticated symmetric encryption scheme SE. The sender then encrypts the plaintext m using the original double ratchet algorithm $\mathsf{ratchet.Enc}(m)$. It bundles the resulting ciphertext ct_m and sender certificate $cert$ and encrypts this with a key derived from long-lived identity keys pk_s and pk_r to produce the content ciphertext ct_{ss}. The sender indicates the intended recipient and sends the triple (pk_e, ct_{id}, ct_{ss}) along with the recipient's access key ak to the platform.

Upon receipt of the sender's message, the platform checks that the intended recipient's registered access key matches ak. If this check passes, then the platform forwards the triple (pk_e, ct_{id}, ct_{ss}) to the recipient. The recipient decrypts as shown in Figure 4.1. Once it recovers $cert$ and ct_m, it verifies the sender as a valid account using the certificate and the recovered identity key pk_s. If the sender's identity is authenticated, then ct_m is decrypted using the double ratchet algorithm.

[1] Signal's use of the double ratchet algorithm produces ciphertexts that can either include the sender identity in plaintext or include messaging metadata such as counters used for in-order processing that would leak information useful for linking senders.

$\mathsf{SealedSender.Send}(m)$	$\mathsf{SealedSender.Rcv}(pk_e, ct_{id}, ct_{ss})$
$ct_m \leftarrow \mathsf{ratchet.Enc}(m)$	$salt_1 \leftarrow (pk_r, pk_e)$
$(pk_e, sk_e) \leftarrow\$ \mathsf{Keygen}()$	$(e_{\mathsf{chain}}, k_e) \leftarrow \mathsf{HKDF}(salt_1, pk_e^{sk_r})$
$salt_1 \leftarrow (pk_r, pk_e)$	$pk_s \leftarrow \mathsf{SE.Dec}(k_e, ct_{id})$
$(e_{\mathsf{chain}}, k_e) \leftarrow \mathsf{HKDF}(salt_1, pk_r^{sk_e})$	$salt_2 \leftarrow (e_{\mathsf{chain}}, ct_{id})$
$ct_{id} \leftarrow\$ \mathsf{SE.Enc}(k_e, pk_s)$	$k \leftarrow \mathsf{HKDF}(salt_2, pk_s^{sk_r})$
$salt_2 \leftarrow (e_{\mathsf{chain}}, ct_{id})$	$cert \| ct_m \leftarrow \mathsf{SE.Dec}(k, ct_{ss})$
$k \leftarrow \mathsf{HKDF}(salt_2, pk_r^{sk_s})$	$b \leftarrow \mathsf{Verify}(pk_s, cert)$
$ct_{ss} \leftarrow\$ \mathsf{SE.Enc}(k, cert \| ct_m)$	If $b = 0$ then return $\perp$
Return $(pk_e, ct_{id}, ct_{ss}), ak$	$m \leftarrow \mathsf{ratchet.Dec}(ct_m)$
	Return m

Figure 4.1: Pseudocode for Signal's sealed sender feature.

4.1.2 Limitations of Sealed Sender

There are limitations to Signal's sealed sender protocol for sender anonymity, which we raise here in the form of three different classes of attacks.

Traffic analysis of sender-anonymous messages. An inherent leakage of the sender-anonymous messaging setting (as opposed to the sender- *and* recipient-anonymous setting) is that the recipient of each message is inherently leaked to the platform. Martiny et al. [MKA+21] demonstrate a set of statistical disclosure attacks that use this leakage to infer communicating partners, for example, by searching for users with interleaving messages suggesting a back-and-forth conversation pattern. They provide a modification to Signal's sealed sender that protects against traffic analysis of sender-anonymous messages, which they call "sender-anonymous conversations". This mitigation approach, as well as another separate approach which instead relies on random message delays and/or noise messages [PHE+17], do not provide solutions for blocklisting. The techniques we introduce for supporting blocklists compose well with these traffic analysis mitigations. Given this prior work, we do not explicitly address traffic analysis of sender-anonymous messages beyond considering the anonymity set, as we discuss next.

Traffic analysis of non-sender-anonymous messages. Recall that access keys are distributed through Signal's original non-sender-anonymous channel. While this setup is still encrypted, the platform nevertheless observes with whom the user exchanged non-sender-anonymous messages. Thus, when

a sender anonymously authenticates using ak, the set of users that could correspond to the sender (i.e., the *anonymity set* of the sender) is restricted and known to the platform. This means, for example, if a recipient only has a single contact with which they have communicated, there is no sender anonymity at all. Furthermore, if a user rotates their access key to revoke sending access, this resets their anonymity set of senders, as their new access key must be redistributed.

Martiny et al. [MKA+21] assume in their threat model that these access keys have already been exchanged between communicating parties. Their attack can therefore be improved by tracking the sender anonymity set of a recipient learned by the platform. Notably, our solution for blocklisting will prevent such improvements.

Griefing attack by evading identification. Sealed sender relies on the sender to self-identify to the recipient: the platform can not check for malformed messages. Instead, the recipient must decrypt and check validity of the sender identity key and certificate, dropping messages that do not verify. This allows for a straightforward griefing attack in which an attacker can spam the recipient with untraceable messages, causing the recipient's device to suffer battery drain and to consume bandwidth, a type of user-mounted DoS attack.

We demonstrate through a proof-of-concept implementation that this griefing attack is effective. Our attack simply modifies pk_e in (pk_e, ct_{id}, ct_{ss}) to a random value pk_f. To the platform this is indistinguishable from a legitimate sealed sender message, but the recipient's decryption will fail when trying to decrypt ct_{id}. The recipient cannot recover any information about the sender.

We performed some measurements to assess whether the griefing attack can be used, particularly, to drain a target's battery. In our experiments, we used as attacker our modified Signal Desktop application on a MacBook Pro 2017 machine running macOS Mojave using a 2.5 GHz Intel Core i7. We used as a stand-in for victim recipient an unmodified Signal Android application (version 4.54.3) on a Google Pixel phone running Android version 9. We used the Android Battery Historian tool to inspect the effect of our attack on battery drainage. It reports the battery level rounded to the nearest percent. We measured the rate of change in battery level per hour when sending one malformed sealed message every 1, 2, 5, or 10 seconds (see Figure 4.2). We find that sending just 1 message every 10 seconds causes the battery to drain at an increased rate of $9\times$ over baseline.

Ultimately, there are no satisfying mitigation options available to victims. If the victim of the

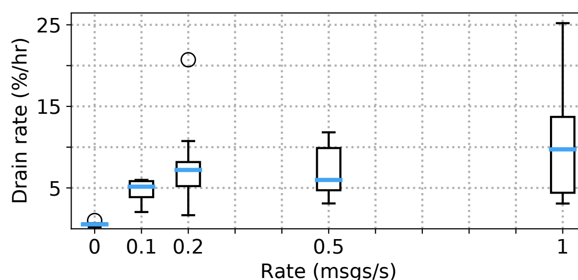

Figure 4.2: Battery drain rate of griefing attack for various rates of sending, $x \in \{0, 0.1, 0.2, 0.5, 1\}$ / second. The box plot shows the variability of drain rates over trials, with the range, quartiles and median denoted by the whiskers, box, and line, respectively (outliers marked separately).

attack has opted in to accepting sealed sender messages from non-contacts, the attack can be mounted by anyone. Otherwise the attacker needs the recipient's access key, meaning the attacker must be one of the victim's contacts (or has found some other way to obtain the access key). While this limits who can mount the attack in the default case, it is still problematic: The victim can rotate their access key ak and attempt to redistribute a new ak' to their communicating partners. If the attacker is not able to get access to the new access key, the attack will be stopped by the platform and no messages will reach the victim's client. But since the attack leaves no information about which of the victim's communicating partners is responsible, the victim can only make a guess as to whom they should block.

Realistically to maintain usability of their mobile device, a user may limit Signal to only a few highly trusted contacts, or will push the user off Signal to a less private messenger. We consider both of these outcomes to be highly damaging to vulnerable users that would benefit from a metadata-private messenger. Looking forward, we will want a mechanism that provides the user more granular recourse against misbehaving senders.

4.2 Outsourced Blocklisting

We now turn to building a new sender-anonymous messaging protocol that avoids the current weaknesses of sealed sender. Our approach is to enable what we call privacy-preserving outsourced blocklisting (see Figure 4.3).

Goals. Such a system should enjoy the following features:

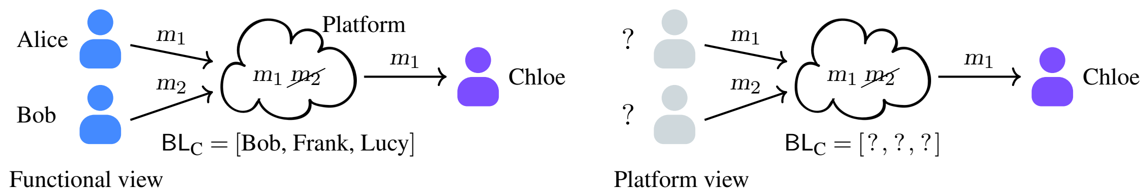

Figure 4.3: Privacy-preserving, outsourced blocklisting for sender-anonymous messaging. The platform is able to block messages from users on Chloe's blocklist without learning their identity. The left view shows the functionality of outsourced blocklisting, while the right view shows what is revealed to the platform. Not shown, Chloe can also efficiently identify the sender of message m_1 as Alice and update her blocklist BL_C if needed.

- *Sender anonymity:* Messages cryptographically hide the sender identity from the platform.
- *Sender attribution:* Recipients can cryptographically verify the sender of any ciphertexts delivered by the platform.
- *Blocklisting:* Recipients can register a blocklist with the platform and update it efficiently. The platform can use the blocklist to drop sender-anonymous messages from senders that the recipient has added to the blocklist.
- *Blocklist anonymity:* The blocklist should not reveal the identities of the senders blocked by the recipient.

Together these properties prevent the type of griefing attacks that affect sealed sender: a client receiving problematic messages can identify the sender and instruct the platform to drop them on the client's behalf.

We would also like the system to support:

- *Non-interactive initialization:* Users can begin sending sender-anonymous messages without previous interaction with the intended recipient.

This property obviates the use of non-sender-anonymous channels to initiate sender-anonymous communication. In particular, the platform should not be able to attribute messages to some smaller subset of users, as messages can have originated from any registered user of the system.

Orca is designed to accompany a sender-anonymous E2EE messaging protocol to provide the functionality of outsourced blocklisting while carrying over both the sender-anonymity and message confidentiality properties of the underlying protocol. As such, we assume the underlying E2EE protocol is sender-anonymous, and if it is not, can easily be made so using encapsulation techniques

similar to sealed sender (see Figure 4.1). Our protocol will provide a registration process in which users interact with the platform to generate the required keys for the protocol; this will be done at the same time users register for the underlying E2EE protocol. To send a message, the sender first encrypts the message plaintext pt to the recipient as specified by the E2EE protocol. Then, Orca will concern itself with authenticating the delivery of the produced E2EE ciphertext; the authenticity of the underlying message plaintext needs to be provided by the E2EE protocol. We will refer to the E2EE ciphertext as the "message" from Orca's perspective.

Threat model. We assume an active, persistent adversary that controls the messaging platform and an arbitrary number of users. We assume the clients of legitimate users are not compromised and that they correctly abide by the protocol.

Our primary concern is the *cryptographic anonymity* of the messaging protocol. The adversary, even with active deviations from the protocol, should not be able to learn sender identity information from the contents of protocol messages.

Even in the case that anonymity is achieved at the message proctocol layer, identification information can leak through the network layer, e.g., by associating IP addresses or by making inferences based on timing. We consider preventing such leakage to be orthogonal to the goal of providing a blocklisting solution for the message protocol layer: existing solutions for mitigating network leakage will compose. Sender-anonymous channels resilient to linking attacks that exploit IP addresses can be constructed using services such as Tor [DMS04]; linking attacks performed by stronger global network adversaries with the ability to observe and inject traffic along any network link can be mitigated using prior academic solutions for anonymous broadcasting [CBM15, WCFJ12, PHE+17, KCDF17, LYK+19]. Lastly, as discussed in Section 4.1.2, given a sender-anonymous channel, timing analysis of messages with designated recipients can be mitigated using existing techniques [PHE+17, MKA+21].

It is trivial for an active adversary that controls the platform to deny service to arbitrary users by not delivering messages. In future work, it may be valuable to provide a mechanism for honest users to provably expose such misbehavior, but in this work we leave platform-mounted denial-of-service (DoS) attacks out of scope. On the other hand, we do want to protect against user-mounted DoS attacks, in which a malicious user can interact with an honest platform to deny service to other users,

as in the griefing attack.

Overview. We will now provide an overview of Orca's design by stepping through a series of strawman constructions.

Sender-specific one-time use access tokens. Instead of having all senders authenticate by reusing the same shared access token, the recipient can deal unique access tokens to each sender. Reusing a sender-specific token allows linking by the platform, so these tokens will necessarily be one-time use only. We outline a version of this approach that is taken by the Pond messaging system [Lan16,LP16].

On registration, recipients register a key k to a pseudorandom function F, e.g. HMAC, with the platform. Recipients distribute one-time use tokens of the form $(x, y = F(k, x))$ for random values x to senders. The platform verifies these tokens using k and the recipient can identify senders since they know to whom they dealt (x, y). A sender's tokens are refreshed in the normal exchange of messages. Now a recipient can block by reporting the unused tokens of a sender to the platform; the platform tracks these tokens along with previously spent tokens for a recipient in a strikelist and rejects incoming messages that authenticate with struck tokens. The platform's strikelist grows unbounded as more messages are sent, but this cost can be managed by scheduled key rotations.

This blocklisting approach improves significantly over sealed sender as it effectively removes the griefing attack vector, however it does not address the concerns around leakage during initialization: the recipient still initially distributes the access tokens over non-sender-anonymous channels to senders, revealing to the platform a small set of possible senders for future messages. A different approach is needed to provide stronger sender anonymity with non-interactive initialization.

Group signatures. A promising starting point for sender-anonymous blocklisting with non-interactive initialization is *group signatures*, a well-studied cryptographic primitive [CvH91, BBS04, Cam98, BMW03, BSZ05]. Group signature schemes allow users to sign messages anonymously on behalf of a group whose membership is controlled by a *group manager*. Signatures appear anonymous to everyone except to a special *opening authority* who has the ability to deanonymize the signer and revoke their signing ability.

Our next strawman solution has the platform maintain a separate group signature scheme for each registered user, where the user is the opening authority and the platform is the group manager. A

sender registers with the platform under the desired recipient's group signature scheme. The sender sends their message along with a signature on the message under the recipient's group to the platform. The platform then verifies the anonymized signature. For blocklisting, we use a group signature scheme that supports *verifier-local revocation* [BS04]. This means that the recipient can revoke senders by communicating *only* with the platform (i.e., verifier).

This strawman provides effective sender attribution and blocklisting. It also allows senders to acquire group signature credentials without previous interaction with the recipient. However, messages to a recipient can be attributed by the platform to the set of users that registered under the recipient's group signature scheme, so we do not achieve our stronger anonymity goal. Furthermore, existing group signatures that meet our requirements use expensive bilinear pairing operations, adding on to the efficiency concerns of managing a separate scheme for each registered user.

We resolve these issues by proposing a new type of group signature that introduces two novel features. The first is support for *multiple opening authorities*. This will dispense with the per-recipient group signature schemes and the need to register separately for each recipient that you wish to send to. The second feature is *keyed-verification*, in which we observe that the platform is also the only verifier. Removing public verifiability improves efficiency of client-side operations.

This new group signature, presented in Section 4.3, makes up the core of Orca. However even with our optimizations, e.g., keyed-verification, the group signature approach incurs significant computational cost, in particular for the platform, owing to the use of verifier-local revocation: verifying a signature incurs work linear in the size of the recipient's blocklist.

Hybrid: Group signature with one-time tokens. This leads us to our final construction which combines the use of group signatures for non-interactive initialization with one-time use tokens for efficient authentication of subsequent messages. Here, the group signature is used to allow the sender to acquire its first batch of tokens from the platform. The main contribution of this approach is a new protocol for allowing the platform to dispense tokens on behalf of the recipient. This is challenging because the platform should not be able to link newly minted tokens to a sender, but it must provide a way for the recipient to learn to whom new tokens were dealt (for future sender attribution). We construct this protocol by adapting techniques from blinded issuance of anonymous credentials [CMZ14]. After this (relatively) expensive initialization procedure, users exchange new

tokens in the normal flow of conversation and the system enjoys all the efficiency benefits of the token-based protocol. We describe Orca's one-time token extension in Section 4.4.

4.3 Blocklisting from Group Signature

Our main construction is based on a novel group signature scheme. In this section, we will introduce our new group signature abstraction, describe how to use it to construct an outsourced blocklisting protocol, and lastly provide an instantiation of such a group signature,

4.3.1 Group Signature Syntax and Security

Group signatures [CvH91] allow users to sign messages anonymously on behalf of a group. The basic setting is as follows. The membership of a group is coordinated by a *group manager*, with whom users register with in order to join the group. Additionally, anonymous group signatures can be opened (traced) to identify the signing user in the group by a designated *opening authority*.

We make use of three extensions to the basic group signature setting.

(1) *Verifier local revocation*: A group signature supporting revocation allows the opening authority to additionally revoke the signing ability of group members. *Verifier local* revocation means that to revoke a member, the opening authority need only communicate a revocation message to verifying parties (as opposed to both verifying parties and group members); revocation does not affect the way group members sign messages.

(2) *Multiple opening authorities*: An opening authority is created through registration with the group manager. Group members sign messages designated to one of many opening authorities, and only the opening authority that a signature is designated to is able to open the signature to the signer's identity. Revocation is handled separately per opening authority, meaning a group member may be able to sign messages designated for some opening authorities, but be revoked from signing messages to others.

(3) *Keyed verification*: Verification of group signatures can only be completed by a secret key owned by the group manager and shared to verifying parties. This is particularly useful in cases where the group manager is the only party verifying signatures and allows for more efficient schemes than those that achieve public verifiability.

Verifier local revocation has been previously studied [BS04], but the other two extensions are novel to the best of our knowledge. The model and following security definitions for our new setting are derived from [BSZ05, BS04].

Syntax

A multi-opener, keyed-verification group signature scheme GS is run between three types of participating parties: (1) users U that join the group and sign messages, (2) opening authorities OA that can trace signatures to signers, and (3) a group manager GM to coordinate registration and perform verification. It consists of the following algorithms:

- $pp \leftarrow\$\ \mathsf{GS.Setup}(\lambda)$: The setup algorithm defines the public parameters pp. We will assume pp is available to all algorithms, and all parties have assurance it was created correctly.
- $(gmpk, gmsk) \leftarrow\$\ \mathsf{GS.Keygen}^{pp}_{\mathrm{GM}}()$: The key generation algorithm is run by the group manager to generate a public key $gmpk$ and secret key $gmsk$.
- $\mathsf{GS.JoinU}^{pp}_{\mathrm{U}} \leftrightarrow \mathsf{GS.IssueU}^{pp}_{\mathrm{GM}}$: Group registration is an interactive protocol implemented by $\mathsf{GS.JoinU}$ and $\mathsf{GS.IssueU}$ run between a user and the group manager, respectively. If execution is successful, the user will receive a public, secret key pair (upk, usk) and the group manager will receive upk, else both parties receive $\perp$. If the protocol accepts, the group manager will store upk in a global registration table and reject duplicate upk registrations.
- $\mathsf{GS.JoinOA}^{pp}_{\mathrm{OA}} \leftrightarrow \mathsf{GS.IssueOA}^{pp}_{\mathrm{GM}}$: Opening authority registration is an interactive protocol run between a prospective opening authority and the group manager. If execution is successful, the opening authority will receive a public, secret key pair $(oapk, oask)$ and the group manager will receive and store $oapk$ in the registration table, else both parties receive $\perp$.
- $\sigma \leftarrow\$\ \mathsf{GS.Sign}^{pp}_{\mathrm{U}}(usk, gmpk, oapk, m)$: The signing algorithm is run by a group member to produce a group signature σ on a message m designated for opening authority $oapk$.
- $upk \leftarrow \mathsf{GS.Open}^{pp}_{\mathrm{OA}}(oask, m, \sigma)$: The opening algorithm is run by an opening authority to learn the identity of the signing user upk, and returns $\perp$ upon failure.
- $\tau_R \leftarrow\$\ \mathsf{GS.Revoke}^{pp}_{\mathrm{OA}}(oask, upk)$: The revocation algorithm is run by an opening authority to create a revocation token τ_R for a user upk. The opening authority sends the revocation token to the group manager who includes it in a revocation list RL used for verification.
- $b \leftarrow \mathsf{GS.Ver}^{pp}_{\mathrm{GM}}(gmsk, oapk, RL, m, \sigma)$: The verification algorithm is run by the group manager

to determine if an input signature σ and m are valid under a designated opening authority $oapk$ and revocation list RL.

As mentioned, we assume some global registration table that contains all user public keys upk and opening authority public keys $oapk$ that succeed registration. In practice, such a table might be implemented with a public key infrastructure (PKI) supporting key transparency audits [MBB+15] allowing it be hosted by the untrusted platform. Additionally, for simplicity, we may drop the executing party from the subscript and the public parameters from the superscript if their use is clear from context.

Correctness and Security Notions

We extend the standard notions of correctness and security from [BSZ05, BS04]. Here, we describe correctness and then the three security properties: anonymity, traceability, and non-frameability. The properties are formalized via security games involving an adversary.

Correctness. The *correctness* property concerns signatures generated by honest group members. An honestly generated signature should pass verification under all honestly generated revocation lists that do not include a revocation token for the signing user created by the designated opening authority. An honestly generated signature should also be opened to the correct signing user by the designated opening authority.

It is defined by the game $\textsc{Corr}$ shown in Figure 4.4 and explained below. We define the advantage of adversary $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \Pr\left[\textsc{Corr}^{\mathcal{A}}_{\mathsf{GS}}(\lambda) = 1\right].$$

We say that a verifier-local revocable, keyed-verification, multi-opener group signature GS is *correct* if $\mathbf{Adv}^{\mathrm{corr}}_{\mathsf{GS},\mathcal{A}}(\lambda) = 0$ for any adversary $\mathcal{A}$ and any $\lambda \in \mathbb{N}$. Note that the adversary is not computationally restricted.

In the correctness game, the adversary can query $\textsc{AddU}$ and $\textsc{AddOA}$ oracles to register new users and opening authorities, each running their respective join/issue interactive protocol with the group manager; the adversary is given the public and secret key of the registered party. The adversary can also query $\textsc{Revoke}$ to add user i to opening authority j's revocation list; the adversary is given the revocation token. After interacting with these oracles, the adversary outputs a msg, user i, and

Game $\text{CORR}^{\mathcal{A}}_{\mathsf{GS}}(\lambda)$	$\text{ADDX}(i)_{X \in \{U,OA\}}$
$pp \leftarrow\$ \mathsf{GS.Setup}(\lambda)$	Require $i \notin H_X$
$(gmpk, gmsk) \leftarrow\$ \mathsf{GS.Keygen}()$	$st \leftarrow \bot$; $st_{GM} \leftarrow \bot$
$(i, j, m) \leftarrow\$ \mathcal{A}^{\text{ADDX},\text{REVOKE}}(gmsk)$	$m_{in} \leftarrow \bot$; $dec \leftarrow \texttt{cont}$
If $i \notin H_U \vee j \notin H_{OA}$ then return 0	While $dec = \texttt{cont}$ do
$(upk, usk) \leftarrow H_U[i]$; $(oapk, oask) \leftarrow H_{OA}[j]$	$(m_{in}, dec) \leftarrow\$ \mathsf{GS.JoinX}(gmpk, m_{in} : st)$
$\sigma \leftarrow\$ \mathsf{GS.Sign}(usk, gmpk, oapk, m)$	$(m_{in}, dec) \leftarrow\$ \mathsf{GS.IssueX}(gmsk, m_{in} : st_{GM})$
$b \leftarrow \mathsf{GS.Ver}(gmsk, oapk, RL[j], m, \sigma)$	If $dec = \texttt{accept}$ then
If $b = 1 \wedge i \in RL_U[j]$ then return 1	$(pk, sk) \leftarrow st$
If $b = 0 \wedge i \notin RL_U[j]$ then return 1	$REG_X[i] \leftarrow pk$; $H_X[i] \leftarrow (pk, sk)$
$upk' \leftarrow\$ \mathsf{GS.Open}(oask, gmpk, m, \sigma)$	$\text{REVOKE}(i, j)$
If $upk \neq upk'$ then return 1	Require $i \in H_U \wedge j \in H_{OA}$
Return 0	$(upk, usk) \leftarrow H_U[i]$; $(oapk, oask) \leftarrow H_{OA}[j]$
	$\tau_R \leftarrow\$ \mathsf{GS.Revoke}(oask, upk)$
	$RL[j] \hookleftarrow \tau_R$; $RL_U[j] \hookleftarrow i$
	Return τ_R

Figure 4.4: Correctness game for keyed-verification multi-opener group signatures.

opening authority j. User i signs message m to opening authority j, and the adversary wins if one of three conditions holds on the signature σ. If the signature verifies with j's revocation list, but user i was on the revocation list from REVOKE, this represents a break of correctness. The second winning condition is the opposite: if the signature does not verify, and user i is not part of the revocation list, that is also incorrect behavior. The last winning condition is if the signature opens to some value other than user i's public key.

Anonymity. The *anonymity* property captures that an adversary without access to the designated opening authority's key should not be able to determine the signer of a signature among unrevoked group members. The adversary has the power of an actively malicious group manager and may adaptively compromise group members and opening authorities. More specifically, we target *CCA-selfless-anonymity* [BBS04] meaning signatures are not anonymous to the signer (selfless) and the adversary has access to an opening oracle throughout the security game (CCA). We consider rogue key attacks, allowing the adversary to create public keys for corrupted parties, but require the adversary to prove knowledge of secret keys. We model this, for simplicity, by asking the adversary to produce the secret key for generated public keys following the knowledge of secret key model

of [Bol03], which can be instantiated with extractible proofs of knowledge. We also provide an extension of our anonymity game to capture anonymity of revocation tokens (in addition to signatures) that is, to our knowledge, the first definitional attempt at doing so.

Anonymity is defined by the game $\textsc{Anon}$ shown in Figure 4.5. We define the advantage of adversary $\mathcal{A}$ as:

$$\mathbf{Adv}^{\text{anon}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \left|\Pr\left[\textsc{Anon}^{\mathcal{A},1}_{\mathsf{GS}}(\lambda) = 1\right] - \Pr\left[\textsc{Anon}^{\mathcal{A},0}_{\mathsf{GS}}(\lambda) = 1\right]\right|.$$

We say that a verifier-local revocable, keyed-verification, multi-opener group signature GS is *anonymous* if $\mathbf{Adv}^{\text{anon}}_{\mathsf{GS},\mathcal{A}}(\cdot)$ is negligible for any polynomial-time adversary $\mathcal{A}$.

In the anonymity game, the adversary plays the role of the platform in attempting to determine the signer's identity of a challenge signature. The adversary may register users and opening authorities using oracles $\textsc{AddU}$ and $\textsc{AddOA}$ (denoted as $\textsc{AddX}$ for $X \in \{U, OA\}$ in the security game) and may corrupt parties to learn their secret key through oracles $\textsc{SKU}$ and $\textsc{SKOA}$ (denoted $\textsc{SKX}$). The adversary can generate signatures for uncorrupted users using $\textsc{Sign}$ and generate revocation tokens from honest opening authorities for arbitrary signatures using $\textsc{OpenRevoke}$. After interacting with these oracles, the adversary may make a single challenge query to $\textsc{ChSign}$ in which they specify two uncorrupted users i_0 and i_1 and an opening authority j and receives a signature from user i_b based on challenge bit b. To disallow trivial wins, neither user's revocation token for j can have been queried via a previous signature to $\textsc{OpenRevoke}$ prior to the challenge query, and are restricted from being queried after the challenge query. The challenge users and opening authority are also restricted from being queried to $\textsc{SKX}$ following the challenge query. The adversary wins if it correctly guesses the challenge bit b.

We extend the game in $\textsc{RevAnon}$ to capture revocation token anonymity (includes highlighted code in Figure 4.5). Here an additional $\textsc{ChRevoke}$ oracle is given to be run on the challenge signature to receive the revocation token for the user i_b. To prevent trivial wins where the adversary holds other signatures from the challenge signers, the $\textsc{ChRevoke}$ oracle rejects queries when either of the two challenge signing users have been queried to $\textsc{Sign}$.

Traceability ensures that every signature that passes verification can be opened by the designated opening authority to a registered user. Traceability necessarily considers an adversary that does not

Game $\text{ANON}^{\mathcal{A},b}_{\mathsf{GS}}(\lambda)$; $\text{REVANON}^{\mathcal{A},b}_{\mathsf{GS}}(\lambda)$

$pp \leftarrow\$ \mathsf{GS.Setup}(\lambda)$
$(gmpk, gmsk) \leftarrow\$ \mathcal{A}(: st_{\mathcal{A}})$
Require $\mathsf{GS.valid}_{\mathrm{GM}}(\lambda, gmpk, gmsk)$
$b' \leftarrow\$ \mathcal{A}^{\text{WREGOA},\text{SIGN},\text{CHSIGN},\text{ADDX},\text{SKU},\text{OPENREVOKE},\text{CHREVOKE}}(: st_{\mathcal{A}})$
Return b'

$\text{WREGOA}(i, pk, sk)$

Require $\mathsf{GS.valid}_{\mathrm{OA}}(\lambda, pk, sk)$
If $i \in H_{OA}$ then $(pk, sk) \leftarrow H_{OA}$
$REG_{OA}[i] \leftarrow (pk, sk)$

$\text{SIGN}(i, j, m)$

Require $i \in H_U \wedge j \in REG_{OA}$
$(upk, usk) \leftarrow H_U[i]$; $(oapk, oask) \leftarrow REG_{OA}[j]$
$\sigma \leftarrow\$ \mathsf{GS.Sign}(usk, gmpk, oapk, m)$
$\Sigma[j][\sigma] \leftarrow i$
Return σ

$\text{CHSIGN}(i_0, i_1, j, m)$

Require $i_0, i_1 \notin K_U \wedge j \notin K_{OA}$
Require $i_0, i_1 \in H_U \wedge j \in H_{OA}$
Require $i_0, i_1 \notin RL[j]$
$(upk_0, usk_0) \leftarrow H_U[i_0]$; $(upk_1, usk_1) \leftarrow H_U[i_1]$
$(oapk, oask) \leftarrow H_{OA}[j]$
$\sigma \leftarrow\$ \mathsf{GS.Sign}(usk_b, gmpk, oapk, m)$
$RQ[j] \hookleftarrow [i_0, i_1]$
$\tilde{\Sigma}[\sigma] \leftarrow (i_0, i_1, j)$
Return σ

$\text{ADDX}(i, m_{in})_{X \in \{U,OA\}}$

Require $i \notin H_X$
$(m_{in}, dec) \leftarrow\$ \mathsf{GS.JoinX}(gmpk, m_{in} : st_X[i])$
If $dec = \texttt{accept}$ then
    $(pk, sk) \leftarrow st_X[i]$; $H_X[i] \leftarrow (pk, sk)$
Return (m_{in}, dec)

$\text{SKX}(i)_{X \in \{U,OA\}}$

Require $i \notin RQ[*] \wedge j \notin RQ$
$K_X \hookleftarrow i$
Return $H_X[i]$

$\text{OPENREVOKE}(m, \sigma, j)$

Require $j \in H_{OA}$
Require $\sigma \notin \tilde{\Sigma}$
Require $\sigma \notin \Sigma[j] \vee \Sigma[j][\sigma] \notin RQ[j]$
$(oapk, oask) \leftarrow H_{OA}[j]$
$upk \leftarrow \mathsf{GS.Open}(oask, gmpk, m, \sigma)$
$\tau_R \leftarrow\$ \mathsf{GS.Revoke}(oask, upk)$
If $\sigma \in \Sigma[j]$ do $RL[j] \hookleftarrow \Sigma[j][\sigma]$
Return τ_R

$\text{CHREVOKE}(\sigma)$

Require $\sigma \in \tilde{\Sigma}$; $(i_0, i_1, j) \leftarrow \tilde{\Sigma}[\sigma]$
Require $i_0, i_1 \notin \Sigma[*][*]$
$(upk, usk) \leftarrow H_U[i_b]$; $(oapk, oask) \leftarrow H_{OA}[j]$
$\tau_R \leftarrow\$ \mathsf{GS.Revoke}(oask, upk)$
Return τ_R

Figure 4.5: Anonymity game for keyed-verification multi-opener group signatures. An extension to the anonymity game is provided to capture anonymity of revocation tokens which includes the highlighted code.

control the group manager since it is trivial for the group manager to craft signatures for unregistered public keys. However, traceability is accompanied by *non-frameability* which ensures that it is not possible to forge a signature that opens to an honest user; non-frameability considers a stronger adversary that controls the group manager as in anonymity.

Traceability is defined by the game TRACE shown in Figure 4.6. We define the advantage of adversary $\mathcal{A}$ as:

$$\mathbf{Adv}^{\text{trace}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \Pr\left[\text{TRACE}^{\mathcal{A}}_{\mathsf{GS}}(\lambda) = 1\right]$$

We say that a verifier-local revocable, keyed-verification, multi-opener group signature GS is *traceable* if $\mathbf{Adv}^{\text{trace}}_{\mathsf{GS},\mathcal{A}}(\cdot)$ is negligible for any polynomial-time adversary $\mathcal{A}$.

$\underline{\text{Game } \textsc{Trace}^{\mathcal{A}}_{\mathsf{GS}}(\lambda)}$

$pp \leftarrow\!\!\$\ \mathsf{GS.Setup}(\lambda)$
$(gmpk, gmsk) \leftarrow\!\!\$\ \mathsf{GS.Keygen}()$
$(j, m, \sigma, L) \leftarrow\!\!\$\ \mathcal{A}^{\textsc{Verify},\textsc{AddX}}(gmpk)$
Require $j \in REG_{OA}$; $(oapk, oask) \leftarrow REG_{OA}[j]$
$b_{ver} \leftarrow \mathsf{GS.Ver}(gmsk, oapk, L, m, \sigma)$
$upk \leftarrow \mathsf{GS.Open}(oask, gmpk, m, \sigma)$
$b_{opn1} \leftarrow upk == \perp \vee upk \notin REG_U$
Return $b_{ver} \wedge b_{opn1}$

$\underline{\textsc{KoskX}(i, sk)_{X \in \{U, OA\}}}$

$P_X[i] \leftarrow sk$

$\underline{\textsc{AddX}(i, m_{in})_{X \in \{U, OA\}}}$

Require $i \notin REG_X$
$(m_{in}, dec) \leftarrow\!\!\$\ \mathsf{GS.IssueX}(gmsk, m_{in} : st_{X,i})$
If $dec = \texttt{accept}$ then
  $pk \leftarrow st_{X,i}$; $sk \leftarrow P_X[i]$
  Require $\mathsf{GS.valid}_X(\lambda, pk, sk)$
  $REG_X[i] \leftarrow (pk, sk)$
Return (m_{in}, dec)

$\underline{\textsc{Verify}(j, m, \sigma, L)}$

Require $j \in REG_{OA}$
$(oapk, oask) \leftarrow REG_{OA}[j]$
$b \leftarrow \mathsf{GS.Ver}(gmsk, oapk, L, m, \sigma)$
Return b

Figure 4.6: Traceability game for keyed-verification multi-opener group signatures.

In the traceability game, the adversary plays the role of a set of malicious users and opening authorities with the goal of creating a message, signature pair that verifies under the honest platform, but fails to open at the recipient. The adversary may register as users and opening authorities using $\textsc{AddX}$. The adversary may verify arbitrary signatures under arbitrary revocation lists using $\textsc{Verify}$. Note that a verify oracle is necessary for the keyed-verification setting. After interacting with these oracles, the adversary outputs a message, signature pair along with a revocation list. The adversary wins if the signature verifies, and the open algorithm fails by either returning $\perp$ or returning an unregistered public key upk.

Non-frameability is defined by the game $\textsc{NFrame}$ shown in Figure 4.7. We define the advantage of adversary $\mathcal{A}$ as:

$$\mathbf{Adv}^{\mathsf{nf}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \Pr\left[\textsc{NFrame}^{\mathcal{A}}_{\mathsf{GS}}(\lambda) = 1\right]$$

We say that a verifier-local revocable, keyed-verification, multi-opener group signature GS is *non-frameable* if $\mathbf{Adv}^{\mathsf{nf}}_{\mathsf{GS},\mathcal{A}}(\cdot)$ is negligible for any polynomial-time adversary $\mathcal{A}$.

The non-frameability game is similar to the traceability game in that the adversary's goal is to output a signature with unwanted opening behavior. However, in the non-frameability game, we consider a stronger adversary that actively controls the platform, similar to the anonymity game. In the non-frameability game, the adversary wins if the signature opens to an honest user not controlled by

Game $\text{NFRAME}_{\mathsf{GS}}^{\mathcal{A}}(\lambda)$

$pp \leftarrow\$ \mathsf{GS.Setup}(\lambda)$
$(gmpk, gmsk) \leftarrow\$ \mathcal{A}(: st_{\mathcal{A}})$
Require $\mathsf{validGS}_{\mathrm{GM}}(\lambda, gmpk, gmsk)$
$(j, m, \sigma) \leftarrow\$ \mathcal{A}^{\text{WREGOA},\text{ADDX},\text{SIGN},\text{OPENREVOKE},\text{SKX}}(: st_{\mathcal{A}})$
Require $j \in H_{OA}$; $(oapk, oask) \leftarrow H_{OA}[j]$
$b_{ver} \leftarrow \mathsf{GS.Ver}(gmsk, oapk, RL[j], m, \sigma)$
$upk \leftarrow \mathsf{GS.Open}(oask, gmpk, m, \sigma)$
$b_{opn2} \leftarrow upk \in H_U \wedge upk \notin K_U$
Return $(upk, m) \notin \mathcal{Q} \wedge b_{ver} \wedge b_{opn2}$

$\text{WREGOA}(i, pk, sk)$

Require $\mathsf{GS.valid}_{\mathrm{OA}}(\lambda, pk, sk)$
If $i \in H_{OA}$ then $(pk, sk) \leftarrow H_{OA}$
$REG_{OA}[i] \leftarrow (pk, sk)$

$\text{SIGN}(i, j, m)$

Require $i \in H_U \wedge j \in REG_{OA}$
$(upk, usk) \leftarrow H_U[i]$; $(oapk, oask) \leftarrow REG_{OA}[j]$
$\sigma \leftarrow\$ \mathsf{GS.Sign}(usk, gmpk, oapk, m)$
$\mathcal{Q} \leftarrow\!\!\shortmid (upk, m)$
Return σ

$\text{ADDX}(i, m_{in})_{X \in \{U, OA\}}$

Require $i \notin H_X$
$(m_{in}, dec) \leftarrow\$ \mathsf{GS.JoinX}(gmpk, m_{in} : st_X[i])$
If $dec = \texttt{accept}$ then
  $(pk, sk) \leftarrow st_X[i]$; $H_X[i] \leftarrow (pk, sk)$
Return (m_{in}, dec)

$\text{SKX}(i)$

$K_X \leftarrow\!\!\shortmid i$
Return $H_X[i]$

$\text{OPENREVOKE}(m, \sigma, j)$

Require $j \in H_{OA}$
$(oapk, oask) \leftarrow H_{OA}[j]$
$upk \leftarrow \mathsf{GS.Open}(oask, gmpk, m, \sigma)$
$\tau_R \leftarrow\$ \mathsf{GS.Revoke}(oask, upk)$
Return τ_R

Figure 4.7: Non-frameability game for keyed-verification multi-opener group signatures.

the adversary, i.e., creates a successful forged signature. The adversary may register honest users and opening authorities using ADDX and may corrupt parties to learn their secret key through SKX. The adversary can generate signatures for uncorrupted users using SIGN and generate revocation tokens on arbitrary signatures using OPENREVOKE. After interacting with these oracles, the adversary outputs a message, signature pair and revocation list. The adversary wins if the message, signature pair was not previously output from SIGN, the signature verifies, and the open algorithm returns the public key of an uncorrupted user. Since the non-frameabilty game captures an adversary with similar power to that of the anonymity game, we make many of the same game design decisions.

4.3.2 Construction of Group Signature

Our group signature follows closely the "certified signature" recipe that many group signatures take [Gro07]. In this recipe, the group manager registers users by certifying their public key $Y = g^y$; the user's group key is made up of their secret identity key y along with the group manager's certificate

t. To sign a message under the group, the user encrypts their public key to the opening authority creating an *identity ciphertext* where Z is the opening authority's encryption key.

$$ct_{id} \leftarrow (g_1^{\alpha_{ct}}, YZ^{\alpha_{ct}}) \qquad \alpha_{ct} \leftarrow\!\!\$\, \mathbb{Z}_p$$

They then prove in zero knowledge that they have a certificate from the group manager on the same public key that is enclosed in the ciphertext *and* that they know the secret key associated to it. The signature is verified by verifying the zero knowledge proof and can be opened by the opening authority simply by decrypting the identity ciphertext.

This recipe naturally extends to support a scheme with multiple opening authorities. The identity ciphertext is encrypted using the public key of the designated opening authority.

Supporting verifier-local revocation. An opening authority registers with two keys: (1) an encryption key $(z, Z = g_1^z)$, and (2) a revocation key $(w, W = g_1^w)$, where $oapk = (W, Z)$. We have described how a user with identity key $(y, Y = g_1^y)$ encrypts their public key Y to the opening authority. To revoke a user's signing ability, the opening authority constructs a user-specific *revocation token* as the Diffie-Hellman value between the user's public key and their own revocation key, $\tau_R = Y^w$. Intuitively, these revocation tokens are anonymous since a Diffie-Hellman value looks random to a verifier that does not know the secret keys y or w.

To allow a verifier in possession of a user's revocation token to identify signatures from a user, we need something more. In addition to the identity ciphertext, the user also constructs a *revocation ciphertext* enclosing their revocation token, $\tau_R = W^y$. This "ciphertext" is constructed to be undecryptable, but includes a backdoor for testing whether a plaintext pt is enclosed (following the approach of Boneh and Shacham [BS04]).

$$ct_R \leftarrow (M_1^{\alpha_T}, \tau_R N_1^{\alpha_T}) \qquad \alpha_T \leftarrow\!\!\$\, \mathbb{Z}_p \qquad M_1, N_1 \leftarrow\!\!\$\, \mathbb{G}_1$$

The backdoor of ct_R consists of the isomorphic $\mathbb{G}_2$ elements M_2, N_2. The verifier can check whether $\hat{\tau}_R$ is enclosed in ct_R via the following test using the pairing function e:

$$e(T_2/\hat{\tau}_R, M_2) \stackrel{?}{=} e(T_1, N_2) \qquad (T_1, T_2) \leftarrow ct_R$$

The verifier performs this test for each revocation token in an opening authority's revocation list and outputs 1 if no revocation token matches and the signature's proof verifies. The signature's proof

now additionally proves the well-formedness of ct_R with respect to user public key Y.

Improving efficiency with keyed-verification. A central part of the group signature is that the user must prove they have a certificate on their public key from the group manager. Creating this proof, even for certificate signatures designed for this purpose [BBS04, CL04], is relatively expensive, with known constructions requiring multiple pairings to be evaluated. In our setting, the platform plays the role of both the group manager and the sole verifier; all messages pass through the platform. This setting allows us to bring in techniques from keyed-verification anonymous credentials [CMZ14]. Specifically, during user registration, instead of receiving a signature from the group manager, users receive a MAC t on their public key from an algebraic MAC scheme; our construction uses $\mathsf{MAC}_{\mathsf{GGM}}$ from [CMZ14, DKPW12]. Proving knowledge of a valid MAC is more efficient and, in particular, does not require pairing evaluations. The resulting proof can only be verified using the secret MAC key (held by the group manager), hence our introduction of the keyed-verification setting for group signatures (i.e., "group MACs"). This optimization limits the use of pairings in our group signature only to the revocation token tests made by the group manager during verification.

Summary. In total, our group signature is composed of three components, (1) the identity ciphertext ct_{id} enclosing the signer's public key to the opening authority, (2) the revocation ciphertext ct_R enclosing the revocation token, and (3) a zero knowledge proof π that (1) and (2) were constructed properly with knowledge of a key pair (y, Y) and a MAC t on Y. The full details of the construction are given in Figure 4.8. Our security proofs are independent of the choice of zero knowledge proof system with which to instantiate the scheme, relying only on the simulation-extractability and zero-knowledge properties described above. In our implementation, we evaluate the classic proof system based on use of Sigma protocols, the building blocks of which are outlined by Camenisch [Cam98]. Our proofs of knowledge are made non-interactive using the Fiat-Shamir heuristic in which the Sigma protocol commitments and proof statement are hashed to get the challenge for the Sigma protocol. The signature of knowledge algorithms are instantiated with the Fiat-Shamir heuristic by additionally passing m into the hash function along with the commitments and statement when generating a challenge. It has been shown that the simulation-extractibility property holds in the algebraic group model [FKL18] for the knowledge of discrete logarithm relation [FPS20, ABM15]. We believe the

techniques used [FPS20] can be applied to show simulation-extractability of the discrete logarithm relations used in this work.

As stated, every time a user sends a message, they create a group signature and the platform verifies the group signature. Even with our optimizations, this involves the platform running a verification algorithm that is linear in the size of the recipient's revocation list. We improve in the next section, extending Orca with one-time use sender tokens to make the need for a group signature a rare event.

4.3.3 Security Analysis

We prove security of our scheme with respect to our formal definitions of anonymity, traceability, and non-frameability. We forgo a formal proof of correctness for our scheme, as it is relatively straightforward to confirm through inspection. First, we provide the following theorems and defer the full proofs of security to the full version [TLMR21]:

Theorem 9. *Let* GS *be the keyed-verification, multi-opener group signature scheme defined in Figure 4.8 over the bilinear group generated by* BGGen*. Let* $\mathsf{MAC}_{\mathsf{GGM}}$ *be the keyed-verification anonymous credentials scheme from [CMZ14] on* $\mathbb{G}_1$*. Let* ElG *be the ElGamal encryption scheme on* $\mathbb{G}_1$*. Then for any adversary* $\mathcal{A}$ *against the anonymity of* GS*, we give adversaries* $\mathcal{A}_1$ *to* $\mathcal{A}_6$ *such that*

$$\mathbf{Adv}^{\mathrm{anon}}_{\mathsf{GS},\mathcal{A}}(\lambda) \leq 2q_u^2 q_{oa}\big(\mathbf{Adv}^{\mathrm{sound}}_{\Pi,\mathcal{A}_1,\mathsf{X}_\Pi}(\lambda) + \mathbf{Adv}^{\mathrm{anon}}_{\mathsf{MAC}_{\mathsf{GGM}},\mathcal{A}_2,\mathsf{S}_{\mathsf{MAC}}}(\lambda) + \mathbf{Adv}^{\mathrm{simext}}_{\Sigma,\mathcal{A}_3,\mathsf{X}_\Sigma,\mathsf{S}_{\mathcal{R}_2}}(\lambda) \\ + \mathbf{Adv}^{\mathrm{indcpa}}_{\mathsf{ElG},\mathcal{A}_4}(\lambda) + 2\cdot\mathbf{Adv}^{\mathrm{ddh}}_{\mathsf{BGGen},\mathcal{A}_5}(\lambda) + \mathbf{Adv}^{\mathrm{dlin}}_{\mathsf{BGGen},\mathcal{A}_6}(\lambda)\big)$$

where $\mathcal{A}$ *makes at most* q_u *and* q_{oa} *queries to the add user and add opening authority oracles, respectively.*

Proof sketch: We bound the advantage of $\mathcal{A}$ by bounding the advantage of each of a series of game hops. We define $\mathrm{G}^b = \textsc{Anon}^{\mathcal{A},b}_{\mathsf{GS}}(\lambda)$ and define games $\mathrm{G}^b_{\mathrm{A}}$, $\mathrm{G}^b_{\mathrm{B}}$, $\mathrm{G}^b_{\mathrm{C}}$, $\mathrm{G}^b_{\mathrm{D}}$, $\mathrm{G}^b_{\mathrm{E}}$, $\mathrm{G}^b_{\mathrm{F0}}$, $\mathrm{G}^b_{\mathrm{F1}}$, G_{G} to gradually transform the view of the adversary until in G_{G} it is no longer dependent on bit b. The inequality above follows from simple calculations based on the following claims which we will justify:

(1) $\mathbf{Adv}^{\mathrm{anon}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \big|\Pr[\mathrm{G}^0 = 1] - \Pr[\mathrm{G}^1 = 1]\big| \leq q_u^2 q_{oa} \cdot \big|\Pr[\mathrm{G}^0_{\mathrm{A}} = 1] - \Pr[\mathrm{G}^1_{\mathrm{A}} = 1]\big|$

1 $\underline{\mathsf{GS.Setup}(\lambda)}$

2 $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,g_1,g_2,e) \leftarrow \mathsf{BGGen}(\lambda)$

3 $\alpha \leftarrow\!\$\, \mathbb{Z}_p$; $h_1 \leftarrow g_1^{\alpha}$

4 $REG_U, REG_{OA} \leftarrow [\cdot]$

5 $pp \leftarrow (p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e,g_1,h_1,g_2)$

6 $pp_{\mathsf{M}} \leftarrow (p,\mathbb{G}_1,g_1,h_1)$

7 Return pp

8 $\underline{\mathsf{GS.JoinU}_{\mathrm{U}}^{pp}(gmpk, m_{in} : st)}$

9 If $m_{in} == \bot$ do

10 $\quad y \leftarrow\!\$\, \mathbb{Z}_p$; $Y \leftarrow g_1^{y}$

11 $\quad m_{out} \leftarrow Y$; $st \leftarrow (Y,y)$

12 $\quad$ Return $(m_{out}, \mathtt{cont}, st)$

13 $(t,\pi,Y_r) \leftarrow m_{in}$

14 $(u_0,u_1) \leftarrow t$; $(y,Y) \leftarrow st$; $(X_1, C_{\tilde{x}_0}) \leftarrow gmpk$

15 $b \leftarrow \Pi.\mathsf{Ver}(\pi,(g_1,h_1,u_0,u_1,X_1,C_{\tilde{x}_0},Y,Y_r))$

16 $usk \leftarrow (t,y)$; $upk \leftarrow Y$

17 $st \leftarrow (upk, usk)$

18 If $b == 1$ then return $(\bot, \mathtt{accept}, st)$

19 Return $(\bot, \mathtt{reject}, st)$

20 $\underline{\mathsf{GS.IssueU}_{\mathrm{GM}}^{pp}(gmsk, m_{in} : st)}$

21 $Y \leftarrow m_{in}$

22 $st \leftarrow Y$

23 $t \leftarrow \underline{\mathsf{MAC}_{\mathsf{GGM}}.\mathsf{GroupElemEv}^{pp_{\mathsf{M}}}(gmsk, Y)}$

24 $\quad (x_0,x_1,\tilde{x}_0) \leftarrow gmsk$

25 $\quad r \leftarrow\!\$\, \mathbb{Z}_p$

26 $\quad u_0 \leftarrow g_1^{r}$; $u_1 \leftarrow (g_1^{x_0} Y^{x_1})^{r}$

27 $\quad$ Return (u_0,u_1)

28 $Y_r \leftarrow Y^{r}$; $X_1 \leftarrow h_1^{x_1}$; $C_{\tilde{x}_0} \leftarrow g_1^{x_0} h_1^{\tilde{x}_0}$

29 $\pi \leftarrow\!\$\, \Pi.\mathsf{Prove}($

30 $\quad (x_0,x_1,\tilde{x}_0,r),(g_1,h_1,u_0,u_1,X_1,C_{\tilde{x}_0},Y,Y_r))$

31 $m_{out} \leftarrow (t,\pi,Y_r)$

32 Return $(m_{out}, \mathtt{accept}, st)$

33 $\underline{\mathsf{GS.JoinOA}_{\mathrm{OA}}^{pp}(gmpk, m_{in} : st)}$

34 If $m_{in} == \bot$ do

35 $\quad w \leftarrow\!\$\, \mathbb{Z}_p$; $W \leftarrow g_1^{w}$

36 $\quad z \leftarrow\!\$\, \mathbb{Z}_p$; $Z \leftarrow g_1^{z}$

37 $\quad m_{out} \leftarrow (W,Z)$; $oapk \leftarrow (W,Z)$; $oask \leftarrow (w,z)$

38 $\quad st \leftarrow (oapk, oask)$

39 $\quad$ Return $(m_{out}, \mathtt{cont}, st)$

40 Return $(\bot, \mathtt{accept}, st)$

41 $\underline{\mathsf{GS.IssueOA}_{\mathrm{GM}}^{pp}(gmsk, m_{in} : st)}$

42 $(W,Z) \leftarrow m_{in}$

43 $st \leftarrow (W,Z)$

44 Return $(\top, \mathtt{accept}, st)$

45 $\underline{\mathsf{GS.Keygen}_{\mathrm{GM}}^{pp}()}$

46 $(gmpk, gmsk) \leftarrow\!\$\, \underline{\mathsf{MAC}_{\mathsf{GGM}}.\mathsf{Keygen}^{pp_{\mathsf{M}}}()}$

47 $\quad x_0 \leftarrow\!\$\, \mathbb{Z}_p$; $x_1 \leftarrow\!\$\, \mathbb{Z}_p$; $\tilde{x}_0 \leftarrow\!\$\, \mathbb{Z}_p$

48 $\quad X_1 \leftarrow h_1^{x_1}$; $C_{\tilde{x}_0} \leftarrow g_1^{x_0} h_1^{\tilde{x}_0}$

49 $\quad pk \leftarrow (X_1, C_{\tilde{x}_0})$; $sk \leftarrow (x_0,x_1,\tilde{x}_0)$

50 $\quad$ Return (pk, sk)

51 Return $(gmpk, gmsk)$

52 $\underline{\mathsf{GS.Sign}_{\mathrm{U}}^{pp}(usk, gmpk, oapk, m)}$

53 $(t,y) \leftarrow usk$; $(u_0,u_1) \leftarrow t$

54 $(X_1, C_{\tilde{x}_0}) \leftarrow gmpk$; $(W,Z) \leftarrow oapk$

55 $\alpha_{ct}, \alpha_u, \alpha_y, \alpha_T, \beta \leftarrow\!\$\, \mathbb{Z}_p^4$

56 $r_m \leftarrow\!\$\, \mathbb{Z}_p$; $r_n \leftarrow\!\$\, \mathbb{Z}_p$

57 $M_1 \leftarrow g_1^{r_m}$; $M_2 \leftarrow g_2^{r_m}$; $N_1 \leftarrow g_1^{r_n}$; $N_2 \leftarrow g_2^{r_n}$

58 $u_0' \leftarrow u_0^{\beta}$; $u_1' \leftarrow u_1^{\beta}$

59 $\tau_R \leftarrow W^{y}$

60 $ct_1 \leftarrow g_1^{\alpha_{ct}}$; $ct_2 \leftarrow g_1^{y} Z^{\alpha_{ct}}$

61 $T_1 \leftarrow M_1^{\alpha_T}$; $T_2 \leftarrow \tau_R N_1^{\alpha_T}$

62 $C_y \leftarrow u_0'^{y} h_1^{\alpha_y}$; $C_u \leftarrow u_1' g_1^{\alpha_u}$; $V \leftarrow g_1^{-\alpha_u} X_1^{\alpha_y}$

63 $\pi \leftarrow\!\$\, \Sigma.\mathsf{Prove}((y,\alpha_y,\alpha_u,\alpha_{ct},\alpha_T,r_m,r_n),$

64 $\qquad (g_1,h_1,u_0',X_1,C_y,V,W,Z,ct_1,ct_2,M_1,M_2,N_1,N_2,T_1,T_2),m)$

65 $\sigma \leftarrow (u_0', C_y, C_u, V, ct_1, ct_2, M_1, M_2, N_1, N_2, T_1, T_2, \pi)$

66 Return σ

67 $\underline{\mathsf{GS.Open}_{\mathrm{OA}}^{pp}(oask, gmpk, m, \sigma)}$

68 $(w,z) \leftarrow oask$; $W \leftarrow g_1^{w}$; $Z \leftarrow g_1^{z}$; $(X_1, C_{\tilde{x}_0}) \leftarrow gmpk$

69 $(u_0, C_y, C_u, V, ct_1, ct_2, M_1, M_2, N_1, N_2, T_1, T_2, \pi) \leftarrow \sigma$

70 Require $\Sigma.\mathsf{Ver}($

71 $\quad (g_1,h_1,u_0,X_1,C_y,V,W,Z,ct_1,ct_2,M_1,M_2,N_1,N_2,T_1,T_2),m,\pi)$

72 $upk \leftarrow ct_2 / ct_1^{z}$

73 Return upk

74 $\underline{\mathsf{GS.Revoke}_{\mathrm{OA}}^{pp}(oask, upk)}$

75 $(w,z) \leftarrow oask$

76 $\tau_R \leftarrow upk^{w}$

77 Return τ_R

78 $\underline{\mathsf{GS.Ver}_{\mathrm{GM}}^{pp}(gmsk, oapk, RL, m, \sigma)}$

79 $(x_0,x_1,\tilde{x}_0) \leftarrow gmsk$; $(W,Z) \leftarrow oapk$

80 $(u_0, C_y, C_u, V, ct_1, ct_2, M_1, M_2, N_1, N_2, T_1, T_2, \pi) \leftarrow \sigma$

81 For $\tau_R \in RL$ do

82 $\quad$ If $e(T_2/\tau_R, M_2) = e(T_1, N_2)$ then return 0

83 $V' \leftarrow u_0^{x_0} C_y^{x_1} / C_u$

84 $X_1 \leftarrow h_1^{x_1}$; $C_{\tilde{x}_0} \leftarrow g_1^{x_0} h_1^{\tilde{x}_0}$

85 $b \leftarrow \Sigma.\mathsf{Ver}($

86 $\quad (g_1,h_1,u_0,X_1,C_y,V',W,Z,ct_1,ct_2,M_1,M_2,N_1,N_2,T_1,T_2),m,\pi)$

87 Return $V == V' \wedge b$

$$\mathsf{R}_1 = \left\{ \left((x_0,x_1,\tilde{x}_0,r),(g_1,h_1,u_0,u_1,X_1,C_{\tilde{x}_0},Y,Y_r) \right) : u_0 = g^{r} \wedge Y_r = Y^{r} \wedge u_1 = u_0^{x_0} Y_r^{x_1} \wedge C_{\tilde{x}_0} = g_1^{x_0} h_1^{\tilde{x}_0} \wedge X_1 = h_1^{x_1} \right\}$$

$$\mathsf{R}_2 = \Big\{ \left((y,\alpha_y,\alpha_u,\alpha_{ct},\alpha_T,r_m,r_n),(g_1,h_1,u_0,X_1,C_y,V,W,Z,ct_1,ct_2,M_1,M_2,N_1,N_2,T_1,T_2) \right) : C_y = u_0^{y} h_1^{\alpha_y} \wedge V = g_1^{-\alpha_u} X_1^{\alpha_y} \wedge ct_1 = g_1^{\alpha_{ct}} \wedge ct_2 = g_1^{y} Z^{\alpha_{ct}} \wedge M_1 = g_1^{r_m} \wedge M_2 = g_2^{r_m} \wedge N_1 = g_1^{r_n} \wedge N_2 = g_2^{r_n} \wedge T_1 = M_1^{\alpha_T} \wedge T_2 = W^{y} N_1^{\alpha_T} \Big\}$$

Figure 4.8: Keyed-verification, multi-opener group signature with verifier-local revocation $\mathsf{GS}[\mathsf{BGGen},\Pi,\Sigma]$ parameterized by a bilinear group, a proof system for R_1 and signature of knowledge for R_2. It makes up the core primitive of Orca.

(2) $|\Pr[\mathrm{G}_{\mathrm{A}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{B}}^b = 1]| = \mathbf{Adv}^{\text{sound}}_{\Pi,\mathcal{A}_1,\mathsf{X}_\Pi}(\lambda)$

(3) $|\Pr[\mathrm{G}_{\mathrm{B}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{C}}^b = 1]| = \mathbf{Adv}^{\text{anon}}_{\mathsf{MAC}_{\mathsf{GGM}},\mathcal{A}_2,\mathsf{S}_{\mathsf{MAC}}}(\lambda)$

(4) $|\Pr[\mathrm{G}_{\mathrm{C}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{D}}^b = 1]| = \mathbf{Adv}^{\text{simext}}_{\Sigma,\mathcal{A}_3,\mathsf{X}_\Sigma,\mathsf{S}_{\mathcal{R}_2}}(\lambda)$

(5) $|\Pr[\mathrm{G}_{\mathrm{D}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{E}}^b = 1]| = \mathbf{Adv}^{\text{indcpa}}_{\mathsf{ElG},\mathcal{A}_4}(\lambda)$

(6) $|\Pr[\mathrm{G}_{\mathrm{E}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{F1}}^b = 1]| = 2 \cdot \mathbf{Adv}^{\text{ddh}}_{\mathsf{BGGen},\mathcal{A}_5}(\lambda)$

(7) $|\Pr[\mathrm{G}_{\mathrm{F1}}^b = 1] - \Pr[\mathrm{G}_{\mathrm{G}} = 1]| = \mathbf{Adv}^{\text{dlin}}_{\mathsf{BGGen},\mathcal{A}_7}(\lambda)$

Recall the group signature is composed of three components: (i) the identity ciphertext ct_{id} enclosing the signer's public key to the opening authority, (ii) the revocation ciphertext ct_R enclosing the revocation token, and (iii) a zero knowledge proof π that (i) and (ii) were constructed properly with knowledge of a key pair (y, Y) and a MAC t on Y. To remove the dependence of signing on challenge bit b, our proof steps through each of these components in sequence. Claims 2 and 3 remove the use of signing key y_b in creating (iii) the zero knowledge proof π of a valid MAC. Claims 4 and 5 remove the use of signing key y_b in encrypting (i) the identity ciphertext. And lastly, claims 6 and 7 remove the use of signing key y_b in constructing (ii) the revocation ciphertext.

Theorem 10. *Let* GS *be the keyed-verification, multi-opener group signature scheme defined in Figure 4.8 over the bilinear group generated by* BGGen*. Let* $\mathsf{MAC}_{\mathsf{GGM}}$ *be the keyed-verification anonymous credentials scheme from [CMZ14] on* $\mathbb{G}_1$*. Then for any adversary* $\mathcal{A}$ *against the traceability of* GS*, we give adversary* $\mathcal{B}$ *such that*

$$\mathbf{Adv}^{\text{trace}}_{\mathsf{GS},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\text{unf}}_{\mathsf{MAC}_{\mathsf{GGM}},\mathcal{B}}(\lambda)$$

where $\mathcal{A}$ *makes at most* q_u *and* q_{ver} *queries to the add user, and verify oracles, respectively, and* $\mathcal{B}$ *makes at most* q_u *and* q_{ver} *queries to its issue and show verify oracles, respectively.*

Proof sketch: We bound the advantage of adversary $\mathcal{A}$ by constructing an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to win the KVAC unforgeability game [CMZ14, Definition 6] whenever $\mathcal{A}$ wins the traceability game. Adversary $\mathcal{B}$ simulates the traceability game for $\mathcal{A}$. The issuer parameters from the KVAC unforgeability game are set as $gmpk$, and the ISSUE and SHOWVERIFY oracles are used to simulate the actions of the group manager in ADDU and VERIFY.

To simulate issuing a signing key in ADDU, $\mathcal{B}$ makes a call to the ISSUE oracle to generate a

MAC t and proof π of wellformedness (lines 23-30 of Figure 4.8). To make a call to ISSUE, $\mathcal{B}$ must send the secret signing key usk. This is fine since $\mathcal{B}$ only needs to properly simulate ADDU if a wellformed secret key has been added via KOSKX, otherwise $\mathcal{B}$ will return $\perp$.

To simulate VERIFY, $\mathcal{B}$ runs its SHOWVERIFY oracle on σ with the following added MAC relation ϕ. The SHOWVERIFY oracle will calculate keyed-verifier values and run the verification procedure for R_2 (lines 83-86 in Figure 4.8). The remainder of VERIFY, i.e. checking against the revocation list, can be run directly by $\mathcal{B}$.

$$\phi(y) = ct_1 = g_1^{\alpha_{ct}} \wedge ct_2 = g_1^y Z^{\alpha_{ct}} \wedge M_1 = g_1^{r_m} \wedge M_2 = g_2^{r_m} \wedge N_1 = g_1^{r_n} \wedge N_2 = g_2^{r_n} \wedge T_1 = M_1^{\alpha_T} \wedge T_2 = W^y N_1^{\alpha_T} .$$

If $\mathcal{A}$ wins the game, then $b_{ver} = 1$ meaning verification passed. This tells us two things. First, open did not return $\perp$, since the only way for open to return $\perp$ is if the signature proof verification fails; this cannot be the case since it is also checked by the verification algorithm. This means that open returned a $upk \notin REG_U$. Second, the signature σ verified under relation $\phi(y)$, where it was claimed that $ct_1 = g_1^{\alpha_{ct}} \wedge ct_2 = YZ^{\alpha_{ct}}$ for some $Y = g_1^y$. However, the call to the open algorithm returned a $upk = Y = ct_2/ct_1^z \notin REG_U$ for all y that credentials were issued for. The signature is then an example of a credential show for which the verification passes but $\phi(y) = 0$ allowing $\mathcal{B}$ to win the KVAC unforgeability game.

Theorem 11. *Let* GS *be the keyed-verification, multi-opener group signature scheme defined in Figure 4.8 over the bilinear group generated by* BGGen*. Let* $\mathsf{MAC_{GGM}}$ *be the keyed-verification anonymous credentials scheme from [CMZ14] on* $\mathbb{G}_1$*. Then for any adversary* $\mathcal{A}$ *against the non-frameability of* GS*, we give adversaries* $\mathcal{A}_1$ *to* $\mathcal{A}_4$ *such that*

$$\mathbf{Adv}^{\mathrm{nf}}_{\mathsf{GS},\mathcal{A}}(\lambda) \leq q_u \big(\mathbf{Adv}^{\mathrm{sound}}_{\Pi,\mathcal{A}_1,\mathsf{X}_\Pi}(\lambda) + \mathbf{Adv}^{\mathrm{anon}}_{\mathsf{MAC_{GGM}},\mathcal{A}_2,\mathsf{S_{MAC}}}(\lambda) + \mathbf{Adv}^{\mathrm{simext}}_{\Sigma,\mathcal{A}_3,\mathsf{X}_\Sigma,\mathsf{S}_{\mathcal{R}_2}}(\lambda) + \mathbf{Adv}^{\mathrm{dl}}_{\mathsf{BGGen},\mathcal{A}_4}(\lambda)\big)$$

where $\mathcal{A}$ *makes at most* q_u *queries to the add user oracle.*

Proof sketch: We bound the advantage of $\mathcal{A}$ by bounding the advantage of each of a series of game hops. Similarly to as in the anonymity proof, we define $\mathrm{G} = \mathrm{NFRAME}^{(}_{\mathsf{GS},\mathcal{A}}\lambda)$ and define games $\mathrm{G_A}$, $\mathrm{G_B}$, $\mathrm{G_C}$, and $\mathrm{G_D}$ that slowly transform the view of the adversary so that signing queries for a guessed user are no longer dependent on their secret key. Then we will show in the final game

$\mathrm{G_D}$, if $\mathcal{A}$ wins, we can win the discrete logarithm game. The inequality above follows from simple calculations based on the following claims which we will justify:

(1) $\mathbf{Adv}^{\mathrm{nf}}_{\mathsf{GS},\mathcal{A}}(\lambda) = \Pr[\mathrm{G}=1] \le q_u \cdot \Pr[\mathrm{G_A}=1]$

(2) $|\Pr[\mathrm{G_A}=1] - \Pr[\mathrm{G_B}=1]| = \mathbf{Adv}^{\mathrm{sound}}_{\Pi,\mathcal{A}_1,\mathsf{X}_\Pi}(\lambda)$

(3) $|\Pr[\mathrm{G_B}=1] - \Pr[\mathrm{G_C}=1]| = \mathbf{Adv}^{\mathrm{anon}}_{\mathsf{MAC}_{\mathsf{GGM}},\mathcal{A}_2,\mathsf{S}_{\mathsf{MAC}}}(\lambda)$

(4) $|\Pr[\mathrm{G_C}=1] - \Pr[\mathrm{G_D}=1]| = \mathbf{Adv}^{\mathrm{simext}}_{\Sigma,\mathcal{A}_3,\mathsf{X}_\Sigma,\mathsf{S}_{\mathcal{R}_2}}(\lambda)$

(5) $\Pr[\mathrm{G_D}=1] = \mathbf{Adv}^{\mathrm{dl}}_{\mathbb{G}_1,p,\mathcal{A}_4}(\lambda)$

Claim 1: Without loss of generality, assume calls to AddX are made with incrementing indices, e.g., $i = 1,2,\ldots,q$. $\mathrm{G_A}$ is the same as G except it guesses the signing party i on which $\mathcal{A}$'s winning signature will open to and aborts if it is incorrect. If $\mathcal{A}$ does not win, or if it wins by opening to a upk that does not belong to user i, then $\mathrm{G_A}$ sets a bad_A flag and aborts. This also means $\mathrm{G_A}$ aborts if party i is queried to SKU since $\mathcal{A}$ cannot win on a corrupted user. By an identical-until-bad argument and the fundamental lemma of game playing [BR06], we have that

$$\Pr[\mathrm{G}=1 \wedge \neg\mathsf{bad}_\mathrm{A}] = \Pr[\mathrm{G_A}=1 \wedge \neg\mathsf{bad}_\mathrm{A}]\,.$$

And since $\mathrm{G_A}$ aborts and outputs 0 when bad_A is set, we have

$$\Pr[\mathrm{G}=1 \wedge \neg\mathsf{bad}_\mathrm{A}] = \Pr[\mathrm{G_A}=1]\,.$$

Then, we have

$$\begin{aligned}\Pr[\mathrm{G_A}=1] &= \Pr[\mathrm{G}=1 \wedge \neg\mathsf{bad}_\mathrm{A}] \\ &= \Pr[\neg\mathsf{bad}_\mathrm{A}] \cdot \Pr[\mathrm{G}=1] \qquad (1)\end{aligned}$$

where (1) holds because the condition to set bad_A is independent of the rest of the game G.

Lastly since the party is guessed at random, the probability that the guess is correct and bad_A is not set is at least

$$\Pr[\neg\mathsf{bad}_\mathrm{A}] \ge \frac{1}{q_u}\,.$$

Claims 2-4: The arguments and game hops for claims 2-4 follow analogously to the same claims in the anonymity proof.

Claim 5: Observe that in $\mathrm{G_D}$, the secret key y of signing user i is not used. Yet to win $\mathrm{G_D}$, the adversary $\mathcal{A}$ must produce a verifying signature that opens to Y. Since the extractor for the signature proof did not fail, we have that it will correctly extract y where $Y = g^y$. We build an adversary $\mathcal{A}_4$ for the discrete logarithm game that wins whenever $\mathcal{A}$ wins by setting the signing user's public key Y to the discrete logarithm challenge element and returning the extracted value y.

4.3.4 Outsourced Blocklisting from Group Signatures

Given a keyed-verification, multi-opener group signature with verifier-local revocation, we build our core protocol, detailed in Figure 4.9. The platform plays the role of the group manager. Users register with the platform as both a user of the group and as an opening authority, receiving keys $(usk_i, oask_i)$. For user i to send a message to user j, assume for now that user i has user j's public keys $(upk_j, oapk_j)$. We will describe how user i obtains these keys shortly.

User i signs their message with usk_i under the group signature scheme designating $oapk_j$ as the opening authority. The platform verifies the anonymous group signature against user j's revocation list, and if it verifies, delivers the message and signature to user j, who can then identify the sender, upk_i, by opening the signature. Users can blocklist a sender upk_i to the platform by generating a revocation token under their opening authority key $oask_j$ and sending it to the platform. Anonymity of the group signature and revocation tokens ensure that the platform does not learn sender identity information from messages or from the blocklist; and traceability and non-frameability ensure recipients will be able to properly attribute received messages to a sender.

To achieve our stronger sender anonymity goal, user i must be able to read the public key information of user j needed to start a conversation *without* revealing their own identity to the platform. Since public key information is not sensitive, the platform can provide unrestricted access to PKI lookups that do not require user authentication. Note that the platform can observe the *number* of lookups to a recipient's public key, but learns no information on which users are making those lookups.

Protocol: Orca Outsourced Blocklisting Protocol

Setup:

(1) Public parameters for the group signature scheme are generated, $pp \leftarrow\$ \mathsf{GS.Setup}(\lambda)$.

(2) The platform initializes its state as the group manager of the group signature scheme.

(a) $(gmpk, gmsk) \leftarrow\$ \mathsf{GS.Keygen}^{pp}_{\mathrm{GM}}()$

(b) $T_U \leftarrow [\cdot]$: Table tracking user public keys.

(c) $T_R \leftarrow [\cdot]$: Table tracking user revocation tokens.

Registration:

(1) User registers with platform to acquire group signature signing key with which to send messages, $\mathsf{JoinUGS}^{pp}_{\mathrm{U}} \leftrightarrow \mathsf{GS.IssueU}^{pp}_{\mathrm{GM}}$. User stores usk and platform stores upk.

(2) User registers as opening authority and generates keys with which to receive messages, $\mathsf{GS.JoinOA}^{pp}_{\mathrm{OA}} \leftrightarrow \mathsf{GS.IssueOA}^{pp}_{\mathrm{GM}}$. User stores $oask$ and platform stores $oapk$.

(3) Platform stores public keys in $T_U[upk] \leftarrow oapk$.

(4) Platform initializes empty revocation token list for user, $T_R[oapk] \leftarrow [\cdot]$.

Sending a message:

(1) [Optional] Sender anonymously requests recipient public key ($oapk$) and/or rate-limited prekeys from platform.

(2) Sender signs message specifying the recipient as the opening authority (with recipient's $oapk$), $\sigma \leftarrow\$ \mathsf{GS.Sign}^{pp}_{\mathrm{U}}(usk, gmpk, oapk, m)$. Sender sends message, signature, and recipient to platform, $(m, \sigma, oapk)$.

(3) Platform checks validity of signature against recipient's revocation list,
$b \leftarrow \mathsf{GS.Ver}^{pp}_{\mathrm{GM}}(gmsk, oapk, T_R[oapk], m, \sigma)$. If $b = 1$, then platform delivers (m, σ) to recipient.

Blocklisting a user:

(1) Recipient generates and sends anonymous revocation token to platform,

(a) $upk \leftarrow \mathsf{GS.Open}^{pp}_{\mathrm{OA}}(oask, m, \sigma)$

(b) $\tau_R \leftarrow\$ \mathsf{GS.Revoke}^{pp}_{\mathrm{OA}}(oask, upk)$

(2) Platform adds revocation token to recipient's blocklist, $T_R[oapk] \leftarrow T_R[oapk] \cup \{\tau_R\}$.

(3) [Optional] Recipient stores identities of blocklisted senders and/or reports sender identity to platform.

Figure 4.9: Core protocol based on group signature.

4.4 Extending Blocklisting with One-time Use Tokens

In this section, we describe how to reduce Orca's reliance on its core group signature protocol. Instead of creating and verifying a group signature for every message sent, the group signature will only be used periodically to mint new batches of one-time use sender tokens from the platform. Messages can be sent, with very little cost, by including a valid token for a recipient. Furthermore, once communication with a recipient has been established, a recipient can replenish a sender's tokens

directly in a return message, avoiding the need to mint more token batches from the platform. The protocol is detailed in Figure 4.10.

Blinded MACs as one-time use tokens. We want that a sender can anonymously mint a batch of tokens for a recipient from the platform. The platform should not be able to link the tokens (when they are spent) to the time of minting. To realize this, we again turn to algebraic MACs used by keyed-verification anonymous credentials [CMZ14]; we use $\mathsf{MAC}_{\mathsf{GGM}}$. Each user generates a MAC secret key $sk \leftarrow (x_0, x_1) \in \mathbb{Z}_p^2$ and sends it to the platform. A valid MAC on input $\nu \in \mathbb{Z}_p$ is of the form,

$$t \leftarrow (u_0, u_1 = u_0^{x_0 + x_1 \nu}) \qquad u_0 \leftarrow\!\$\, \mathbb{G}_1 \,.$$

To blindly evaluate a MAC on input ν, a user generates a random ElGamal key pair $(\gamma, D = g_1^{\gamma})$ and encrypts g_1^{ν} to D,

$$ct = (ct_1 = g_1^r, ct_2 = g_1^{\nu} D^r) \qquad r \leftarrow\!\$\, \mathbb{Z}_p \,.$$

The user blinds a batch of inputs $[\nu]_i$ in this manner, creates a group signature σ over $[ct]_i$ designating the recipient as the opening authority, and then sends $(\sigma, [ct]_i, D)$ to the platform. The platform verifies the group signature under the recipient's revocation list, and if verification succeeds, proceeds with the blind evaluation using the recipient's MAC secret key. By the homomorphic properties of ElGamal, the platform can maul ct to form ct' as an encryption of a valid MAC on ν without ever learning anything about ν,

$$ct' = (ct_1^{x_1 \cdot b} g_1^{r'}, ct_2^{x_1 \cdot b} u_0^{x_0} D^{r'}) \quad u_0 \leftarrow g_1^b \qquad b, r' \leftarrow\!\$\, \mathbb{Z}_p \,.$$

The full details of the blind MAC evaluation is given in Figure 4.11. The user decrypts ct' to learn u_1 and stores token $\tau \leftarrow (\nu, t = (u_0, u_1))$ as the input, tag pair.

To send a message, the user sends the message to the platform along with an unused token τ for the recipient. The platform checks that the token $(\nu, t) \leftarrow \tau$ is unused, i.e., ν is not in the strikelist of used tokens for a recipient, and that the token is valid, i.e., the MAC t is valid for ν under the recipient's MAC key. If those checks pass, the platform delivers the message along with the token τ to the recipient and adds ν to the recipient's strikelist.

However, the recipient has no way identifying the sender from the token τ. The generation of τ

was (necessarily) blinded to prevent linking by the platform, but that also prevents linking by the recipient.

Allowing a recipient to link tokens to senders. Senders must communicate to the recipient the unblinded inputs ν for which they are minting tokens. They do this by additionally encrypting the input ν to the recipient under the recipient's public key Z,

$$\hat{ct} = (\hat{ct}_1 = g_1^{\hat{r}}, \hat{ct}_2 = g_1^{\nu} Z^{\hat{r}}) \qquad \hat{r} \leftarrow\!\!\$\, \mathbb{Z}_p\,,$$

and proving in zero knowledge that the input ν enclosed in the blinded ciphertext ct is the same as that enclosed in the ciphertext $\hat{ct}$ to the recipient (details highlighted in Figure 4.11). The sender signs the batch of recipient ciphertexts $[\hat{ct}]_i$ under the group signature with the recipient as the designated opening authority. As before, if the signature σ verifies under the recipient's revocation list, the platform proceeds with blind evaluation, *but also* sends $(\sigma, [\hat{ct}]_i)$ to the recipient.

The recipient opens σ to the sender's identity upk, then decrypts and stores the token identifiers $[g_1^{\nu}]_i$. Later when a recipient receives a message and token $(\nu, t) \leftarrow \tau$ from the platform, they can link the token to the sender by looking up g_1^{ν}. To block a sender, the recipient generates and sends the revocation token for the sender's upk to the platform so the sender cannot mint new tokens, as well as sends the sender's remaining unused tokens $[g_1^{\nu}]_i$ to add to the strikelist.

Replenishing tokens directly from the recipient. The motivation for one-time use tokens was to avoid the cost of the more expensive group signature for every message. However, in some sense, the gain from not running the group signature for every message is offset by the upfront cost of generating a proof to mint each token. While there are optimizations that can be made when batching proofs in this manner [HG13], this is still an unsatisfying result.

The real efficiency gain from one-time use tokens is when senders can replenish their tokens directly from the recipient, without going through the blind minting process with the platform. Once two users have established sender-anonymous communication, they can use their own secret MAC keys to generate and exchange tokens directly at very little cost.

Summary. In this protocol, the core group signature is used only to initiate conversations and mint the first batch of tokens. Once conversation has been established, messages can be exchanged and

Setup:

(1) Public parameters for the group signature scheme, algebraic MAC scheme, and public key encryption scheme are generated, $pp \leftarrow\$ \mathsf{GS.Setup}(\lambda)$, $pp_{\mathsf{M}} \leftarrow\$ \mathsf{MAC.Setup}(\lambda)$, $pp_{\mathsf{PKE}} \leftarrow\$ \mathsf{PKE.Setup}(\lambda)$.

(2) The platform initializes its state as the group manager of the group signature scheme.

(a) $(gmpk, gmsk) \leftarrow\$ \mathsf{GS.Keygen}^{pp}_{\mathrm{GM}}()$

(b) $T_U \leftarrow [\cdot]$: Table storing user public keys.

(c) $T_R \leftarrow [\cdot]$: Table storing user revocation tokens.

(d) $T_k \leftarrow [\cdot]$: Table storing user token MAC key and encryption key.

(e) $T_\tau \leftarrow [\cdot]$: Table storing strikelist of previously-used tokens for user.

Registration:

(1) User generates keys for protocol and initializes recipient state:

(a) User registers with platform to acquire group signature signing key with which to send messages, $\mathsf{GS.JoinU}^{pp}_{\mathrm{U}} \leftrightarrow \mathsf{GS.IssueU}^{pp}_{\mathrm{GM}}$. User stores usk and platform stores upk.

(b) User registers as opening authority and generates keys with which to block senders, $\mathsf{GS.JoinOA}^{pp}_{\mathrm{OA}} \leftrightarrow \mathsf{GS.IssueOA}^{pp}_{\mathrm{GM}}$.

(c) User generates algebraic MAC key used for creating sender tokens, $(tsk, tpk) \leftarrow\$ \mathsf{MAC.Keygen}^{pp_{\mathsf{M}}}()$, and sends both tsk and tpk to platform.

(d) User generates keys for public key encryption scheme, $(ek, dk) \leftarrow\$ \mathsf{PKE.Keygen}()$, stores dk and sends ek to platform.

(e) User initializes two tables, T_x and T_x^{-1}, to identify (and blocklist) senders and their associated sender tokens.

(2) Platform stores keys and initializes table entries for user:
$T_U[upk] \leftarrow (oapk)$; $T_k[oapk] \leftarrow (tsk, tpk, ek)$
$T_R[oapk] \leftarrow [\cdot]$; $T_\tau[oapk] \leftarrow [\cdot]$

Sending a message:

(1) Sender selects unused sender token for recipient and sends message, token, and recipient, $(m, \tau, oapk)$, to platform.

(2) Platform checks if token $(x,t) \leftarrow \tau$ is valid under recipient's MAC key $(tsk, tpk, ek) \leftarrow T_k[oapk]$ and if token was not already used (i.e., is not on strikelist).
$b_1 \leftarrow \mathsf{MAC.Ver}^{pp_{\mathsf{M}}}(tsk, x, t)$
$b_2 \leftarrow (x \notin T_\tau[oapk])$
If $b_1 = 0$ or $b_2 = 0$, platform aborts.

(3) Platform adds token to strikelist, $T_\tau[oapk] \leftarrow T_\tau[oapk] \cup \{x\}$.

(4) Platform forwards message and token value, (m, x), to recipient.

(5) Recipient removes token from list of valid tokens for sender,
$T_x[T_x^{-1}[x]] \leftarrow T_x[T_x^{-1}[x]] \setminus \{x\}$; $T_x^{-1}[x] \leftarrow \perp$.

Acquiring sender tokens (from platform):

(1) [Optional] Sender anonymously requests public key information, $(oapk, tpk, ek)$, for desired recipient from platform.

(2) Sender authenticates to platform as a non-blocklisted sender for the recipient using a group signature.

(a) Sender signs set of recipient ciphertexts $[\hat{ct}]_i$ (constructed in (3)) with recipient as opening authority, and sends $(\sigma, oapk)$ to platform, $\sigma \leftarrow\$ \mathsf{GS.Sign}^{pp}_{\mathrm{U}}(usk, gmpk, oapk, [\hat{ct}]_i)$.

(b) Platform checks validity of signature against recipient's revocation list, $b \leftarrow \mathsf{GS.Ver}^{pp}_{\mathrm{GM}}(gmsk, oapk, T_R[oapk], [\hat{ct}]_i, \sigma)$. If $b = 0$, then platform aborts.

(3) Sender engages in token generation protocol with platform.

(a) Sender samples m inputs, $[x]_i^m \leftarrow\$ \mathsf{MAC.In}(\lambda)^m$.

(b) Sender encrypts inputs to recipient, $\hat{ct}_i \leftarrow\$ \mathsf{PKE.Enc}(ek, x_i)$.

(c) Sender and platform engage in MAC blind evaluation for each token, $\mathsf{MAC.BlindInp}^{pp_{\mathsf{M}}}(tpk, x_i) \leftrightarrow \mathsf{MAC.BlindEv}^{pp_{\mathsf{M}}}(tsk)$, for recipient keys $(tsk, tpk, ek) \leftarrow T_k[oapk]$. Sender also sends proof that the input used in the MAC protocol is properly well-encrypted in the ciphertext to the recipient:
$$\pi_i \leftarrow\$ \Pi\{x_i : \mathsf{MAC.BlindInp}^{pp_{\mathsf{M}}}(tpk, x_i) \wedge ct_i = \mathsf{PKE.Enc}^{pp_{\mathsf{PKE}}}(ek, x_i)\}$$
If π_i does not verify, platform aborts the blind MAC protocol.

(d) If blind MAC protocol succeeds, sender receives MAC t_i as output and stores token, $\tau_i \leftarrow (x_i, t_i)$.

(4) Platform sends $(\sigma, [\hat{ct}]_i^m)$ to recipient.

(5) Recipient stores tokens to later identify sender.

(a) Recipient traces sender, $upk \leftarrow \mathsf{GS.Open}^{pp}_{\mathrm{OA}}(oask, [\hat{ct}]_i, \sigma)$.

(b) Recipient decrypts token ciphertexts and stores tokens.
$x_i \leftarrow \mathsf{PKE.Dec}^{pp_{\mathsf{PKE}}}(dk, \hat{ct}_i)$
$T_x[upk] \leftarrow T_x[upk] \cup [x_1, \ldots, x_m]$; $T_x^{-1}[x_i] \leftarrow upk$

Acquiring sender tokens (from recipient):

(1) Recipient samples m inputs, $(x_1, \ldots, x_m) \leftarrow\$ \mathsf{MAC.In}(\lambda)^m$, and MACs them, $t_i \leftarrow \mathsf{MAC.Ev}^{pp_{\mathsf{M}}}(tsk, x_i)$.

(2) Recipient sends tokens $\tau_i \leftarrow (x_i, t_i)$ to sender associated with upk out-of-band or via secure channel.

(3) Recipient stores tokens to later identify sender.
$T_x[upk] \leftarrow T_x[upk] \cup \{x_1, \ldots, x_m\}$; $T_x^{-1}[x_i] \leftarrow upk$

Blocklisting a user:

(1) Recipient looks up sender identity associated with token, $upk \leftarrow T_x^{-1}[x]$, and generates revocation token, $\tau_R \leftarrow\$ \mathsf{GS.Revoke}^{pp}_{\mathrm{OA}}(oask, upk)$. Recipient sends revocation token along with list of remaining sender tokens for sender to platform, $(x_1, \ldots, x_m) \leftarrow T_x[upk]$.

(2) Platform updates blocklist state by adding revocation token to blocklist and remaining tokens to strikelist.
$T_R[oapk] \leftarrow T_R[oapk] \cup \{\tau_R\}$
$T_\tau[oapk] \leftarrow T_\tau[oapk] \cup \{x_1, \ldots, x_m\}$

Figure 4.10: Hybrid protocol based on group signature and tokens.

$\mathsf{MAC_{GGM}.BlindInp}^{pp_\mathsf{M}}(pk, x, oapk, m_{in} : st)$	$\mathsf{BlindEvMAC_{GGM}}^{pp_\mathsf{M}}(sk, oapk, m_{in} : st)$
$(X_1, C_{\tilde{x}_0}) \leftarrow pk$; $(W, Z) \leftarrow oapk$	$(x_0, x_1, \tilde{x}_0) \leftarrow sk$; $X_1 \leftarrow h_1^{x_1}$; $C_{\tilde{x}_0} \leftarrow g_1^{x_0} h_1^{\tilde{x}_0}$; $(W, Z) \leftarrow oapk$
If $m_{in} == \bot$ do	$(D, ct_1, ct_2, \hat{ct}_1, \hat{ct}_2, \pi) \leftarrow m_{in}$
$\quad (\gamma, r) \leftarrow\$ \mathbb{Z}_p^2$; $D \leftarrow g_1^\gamma$	$b \leftarrow \mathsf{\Pi}_3.\mathsf{Ver}((g_1, D, Z, ct_1, ct_2, \hat{ct}_1, \hat{ct}_2), \pi)$
$\quad ct_1 \leftarrow g_1^r$; $ct_2 \leftarrow g_1^x D^r$	If $b == 0$ then return $(\bot, \mathtt{reject}, st)$
$\quad \hat{r} \leftarrow\$ \mathbb{Z}_p$; $\hat{ct}_1 \leftarrow g_1^{\hat{r}}$; $\hat{ct}_2 \leftarrow g_1^x Z^{\hat{r}}$	$b \leftarrow\$ \mathbb{Z}_p$; $r' \leftarrow\$ \mathbb{Z}_p$; $b_1 \leftarrow x_1 \cdot b$
$\quad \pi \leftarrow \mathsf{\Pi}_3.\mathsf{Prove}((x, r, \hat{r}), (g_1, D, Z, ct_1, ct_2, \hat{ct}_1, \hat{ct}_2))$	$u_0 \leftarrow g_1^b$; $X_b \leftarrow X_1^b$
$\quad m_{out} \leftarrow (D, ct_1, ct_2, \hat{ct}_1, \hat{ct}_2, \pi)$	$ct_1' \leftarrow ct_1^{b_1} g_1^{r'}$; $ct_2' \leftarrow ct_2^{b_1} u_0^{x_0} D^{r'}$
$\quad st \leftarrow (\gamma, ct_1, ct_2)$	$\pi \leftarrow\$ \mathsf{\Pi}_4.\mathsf{Prove}((x_0, x_1, \tilde{x}_0, r', b, b_1),$
$\quad$ Return $(m_{out}, \mathtt{cont}, st)$	$\quad (g_1, h_1, X_1, X_b, C_{\tilde{x}_0}, D, u_0, ct_1, ct_2, ct_1', ct_2'))$
$(ct_1', ct_2', u_0, X_b, \pi) \leftarrow m_{in}$	$m_{out} \leftarrow (ct_1', ct_2', u_0, X_b, \pi)$
$(\gamma, ct_1, ct_2) \leftarrow st$	$st \leftarrow (\hat{ct}_1, \hat{ct}_2)$
$b \leftarrow \mathsf{\Pi}_4.\mathsf{Ver}((g_1, h_1, X_1, X_b, C_{\tilde{x}_0}, g_1^\gamma, u_0, ct_1, ct_2, ct_1', ct_2'), \pi)$	Return $(m_{out}, \mathtt{accept}, st)$
If $b == 0$ then return $(\bot, \mathtt{reject}, st)$	
$u_1 \leftarrow ct_2' / ct_1'^{\gamma}$; $t \leftarrow (u_0, u_1)$; $st \leftarrow t$	
Return $(\bot, \mathtt{accept}, st)$	

$$\mathsf{R}_3 = \left\{ \left((x, r, \hat{r}), (g_1, D, Z, ct_1, ct_2, \hat{ct}_1, \hat{ct}_2) \right) : ct_1 = g_1^r \wedge ct_2 = g_1^x D^r \wedge \hat{ct}_1 = g_1^{\hat{r}} \wedge \hat{ct}_2 = g_1^x Z^{\hat{r}} \right\}$$

$$\mathsf{R}_4 = \left\{ \left((x_0, x_1, \tilde{x}_0, r', b, b_1), (g_1, h_1, X_1, X_b, C_{\tilde{x}_0}, D, u_0, ct_1, ct_2, ct_1', ct_2') \right) : C_{\tilde{x}_0} = g_1^{x_0} h_1^{\tilde{x}_0} \wedge X_1 = h_1^{x_1} \wedge X_b = X_1^b \wedge X_b = h_1^{b_1} \wedge u_0 = g_1^b \wedge ct_1' = ct_1^{b_1} g_1^{r'} \wedge ct_2' = ct_2^{b_1} u_0^{x_0} D^{r'} \right\}$$

Figure 4.11: Modified blind evaluation of algebraic MACs for token generation used in the extension of Orca with one-time tokens using proof systems $\mathsf{\Pi}_3$ and $\mathsf{\Pi}_4$ for relations R_3 and R_4, respectively.

tokens can be replenished at almost no cost, beyond storage. With regards to storage, users must maintain lists of unused tokens in order to send messages and identify senders of received messages. The platform also needs to maintain an ever-growing strikelist for each user; in practice, users will need to periodically rotate their keys to refresh the platform strikelist, but can ensure that they have distributed tokens for the new key prior to doing so.

Using tokens does leak some information about user communication patterns in a nuanced way. An example might be that if senders need to often mint tokens from the platform for a particular user, the platform can infer that user is not active in responding and replenishing sender tokens.

A second nuance is that in both our scheme and the token strawman [Lan16, LP16] presented in Section 4.1, the message ciphertext of a sender is not bound to the token. The platform can forward the sender's token to the recipient, but swap out the ciphertext, so the recipient will incorrectly attribute it to the sender. The impact of such an attack is not large if the underlying E2EE protocol provides message authentication.

Despite these nuances, we feel Orca with one-time use tokens represents an attractive design

Operation		Platform		User (Desktop client)				User (Mobile client)			
				Sender		Recipient		Sender		Recipient	
Sealed sender		–		0.50	(0.02)	0.50	(0.02)	6.6	(0.2)	6.6	(0.2)
Orca	mint tokens with group signature	11.2	(0.2)	10.8	(0.1)	9.7	(0.2)	131.7	(0.8)	117	(2)
	+ cost per token minted	7.60	(0.09)	8.50	(0.08)	0.30	(0.01)	105.2	(0.9)	3.3	(0.1)
	+ cost per blocked user	1.70	(0.04)	–		–		–		–	
	send message with token*	0.30	(0.01)	0.80	(0.02)	–		10.0	(0.2)	–	

*Steady-state cost of sending a message with a token that includes cost of replenishing one token

Figure 4.12: Processing time (ms) microbenchmarks of user and platform operations for Orca compared to sealed sender. Mean time is given with standard deviations shown in parentheses. Dashes indicate an operation that has negligible cost (e.g., a table lookup).

choice.

4.5 Evaluation

This section aims to evaluate the feasibility of deploying Orca at scale. Specifically, we answer the following questions:

- *Client costs*: What are the processing and storage costs that Orca incurs on user clients?
- *Platform costs*: What are the processing and storage costs incurred on the platform? What throughput (user activity) can be reasonably supported given these costs?
- *Bandwidth costs*: How large are Orca protocol messages? What additional networking costs does Orca introduce?

To answer these questions, we provide a prototype library in Rust of our group signature and token-based scheme. Our implementation is over the BLS12-381 pairing-friendly elliptic curve and uses the `zexe/algebra` Rust pairing library [BCG+20a]. We instantiate the proofs of knowledge using standard Sigma protocols of discrete logarithm relations [Cam98] made non-interactive using the Fiat-Shamir transform [FS86]. Our implementation consists of less than 1400 lines of code and is available open source at `https://github.com/nirvantyagi/orca`.

The experiments, including the microbenchmarks given in Figure 4.12, were performed using a `c5.12xlarge` Amazon EC2 virtual machine with 24 cores and 96 GB of memory running Ubuntu Server 20.04 LTS as the platform and desktop client (single-core) and on a Google Pixel device running Android 9 as the mobile client. The platform is implemented using an in-memory Redis database for storing revocation blocklists and token strikelists.

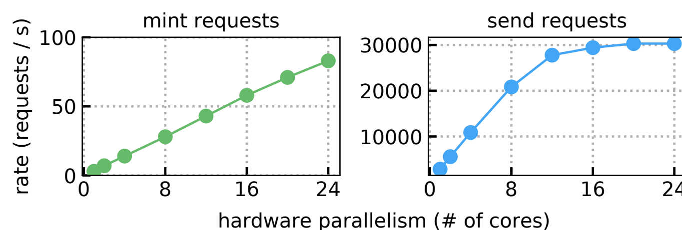

Figure 4.13: Platform request throughput for different levels of hardware parallelism over a one million user deployment with blocklists of size 100 and strikelists of size 1400. Each mint request corresponds to a request to mint a batch of 10 tokens.

When evaluating Orca, recall that users can replenish their token supply directly from the recipient provided there is back and forth communication. Thus, we make the distinction between "initialization costs" of minting an initial token batch from the platform and the "steady-state costs" that occur when tokens are replenished directly from the communicating partner. We expect the majority of user communication to be in steady-state where costs are low.

Client costs. Clients must store, for each of their communicating partners, two lists of unused tokens, one for sending messages and one for identifying received messages. These tokens are not large (240B) and the lists can remain small as they can be replenished on next communication. Say a user has 200 communication partners and stores 20 tokens per list. This setup would incur $\sim$ 1MB for the client.

The bulk of the processing costs incurred by Orca are concentrated at initialization when a client mints an initial batch of tokens to start a conversation. On a mobile client, minting an initial batch of tokens takes $\sim$ 150 ms for the group signature and an additional $\sim$ 100 ms for each token in the batch (see Figure 4.12). This means it takes around 1 second for a sender to mint 10 tokens. While these costs are significant, we stress that a user only needs to mint enough tokens to initiate a conversation and await a response. If a response from a recipient is delayed, more tokens can be minted as needed. Once a conversation with back-and-forth communication is established, the amortized steady-state cost of sending a message is in creating a new token to replenish the recipient, which is done at very little cost ($\sim$ 10 ms) — approximately the same as sealed sender.

Platform costs. The platform stores per-recipient revocation blocklists and token strikelists. The revocation lists are on the order of 100B / revoked user; e.g., a recipient that has blocked 100 users

would require a revocation list of size 10KB to be stored. We do not anticipate revocation lists to grow too large, since the platform has other mechanisms to ban users globally. In any case, a platform can impose limits on the size of revocation lists if necessary.

The per-recipient strikelists would grow in size with every message a user sends (32B / spent token). One can use Bloom filters or other data structures to compress the size of the strikelist as well as enforce periodic key rotations to reset its size. If each user sends ~ 100 messages per day and token keys are rotated every two weeks, the platform can store a strikelist of ~ 5KB per user with a false positive rate of 10^{-6}. Note the false positive rate can be traded off with storage size; messages that get rejected due to false positives will result in an error returned to the anonymous sender, who may resend with a different token.

The processing costs of the platform are similarly dominated by the token mint requests for initializing conversation as opposed to send requests during steady-state conversation. A request to mint a batch of 10 tokens given a recipient blocklist size of 100 takes ~ 200 ms to complete whereas a send request is just a simple algebraic MAC verification and strikelist lookup taking < 1 ms (see Figure 4.12).

Figure 4.13 demonstrates these workloads are easily parallelizable to achieve high levels of throughput. In this experiment, we run the platform with one million users, each with a blocklist of size 100 and a strikelist of size 1400 (100 messages/day/two weeks), and measure the rate at which the platform can process requests for different levels of hardware parallelism. We do not implement the Bloom filter optimization, so the Redis database stores ~ 50KB per user (50GB total), which can still easily fit in memory. The computationally expensive mint requests parallelize with essentially no loss, reaching a rate of 80 requests (for 10 token batches) per second on 24 cores. The inexpensive send requests also parallelize but top out at around 30000 requests per second on 12 cores, which is bottlenecked by the operation throughput of a single Redis database and can be unblocked via a different database setup if needed (e.g. through sharding). The achieved bottlenecked throughput already demonstrates feasibility.

Bandwidth costs. Minting a token requires sending the group signature (1.6KB) and exchanging proofs for each token to be minted (0.7KB / token). These costs extend to the recipient who receives the signature and also a ciphertext for each token minted (0.2KB / token). Apart from these initial-

ization costs, the steady-state bandwidth costs of sending a message, once again, compare quite favorably with sealed sender. In the steady state, the amortized bandwidth overhead of sending a message would be two tokens (240B / token) — the token being spent *and* the token being created to replenish the recipient. Thus we can achieve amortized per-message overheads of only 30B compared to sealed sender (450B / message).

CHAPTER 5

VERIFIABLE PUBLIC KEY INFRASTRUCTURE

A number of systems have demonstrated the promise of *transparency* as a means to enhance security, most prominently the Certificate Transparency protocol first launched in 2013 [LLK13, Lau14]. The goal of transparency systems is to ensure that an authority's behavior can be monitored by users. Typically, misbehavior by the authority is not prevented but is detectable. The implicit assumption is that large, public-facing authorities are potentially malicious (or compromised) but are cautious: they are unwilling or at least extremely hesitant to carry out any attack that will leave public evidence.

Transparency has been proposed in a number of other security contexts, including user-public key mappings in encrypted communication systems [Rya14, MBB+15], usage of cryptographic keys [YRC15], and distribution of software binaries [FDP+14, NKJ+17, AM18]. *Verifiable registries* [CDGM19, MKL+20] are an abstraction capable of providing transparency for the key-value mappings required for all of these applications. Without such a transparency solution, the only defense against malicious behavior by the authority (or *provider*) is out-of-band cross-checking of the authority's behavior (e.g. checking the fingerprints of downloaded public keys), an error-prone process which the vast majority of users neither understand nor attempt [DSB+16, TBB+17, VWO+17, ASB+17].

Client monitoring and auditing. Verifiable registries provide *lookup proofs* (or *binding proofs*) that prove the results of a lookup are consistent with the committed state of the registry at a particular epoch. These lookup proofs can be *monitored* by users to detect any unexpected changes. Typically there is no well-defined notion of correctness for a specific registry mapping as the authority is trusted to update mappings when needed (e.g. account recovery for a user who has lost their private key). Thus, monitoring is inherently a process specific to each mapping and/or user.

By contrast, *auditing* is the process of ensuring that the entire registry is well-formed and maintains promised invariants across epochs. Unlike monitoring, auditing can be fully automated, with any violation by the provider producing unambiguous cryptographic proof of misbehavior. Early constructions propose clients directly perform audits in every epoch [LLK13, Rya14, MBB+15]. As this approach incurs large overhead which is linear in the number of epochs, later proposals instead suggest outsourcing auditing of global registry invariants (such as update counts) to a third party.

This enables clients to monitor their own key-value mapping at a lower frequency, with significant cost savings [MBB+15, CDGM19, MKL+20, TBP+19, HHK+21].

However, this assumes suitable trusted parties exist which can regularly perform expensive global audits. One could rely on the validation process underlying existing blockchain infrastructure (in particular, by implementing auditing in a smart contract [Bon16]), but this may result in large transaction fees. In this paper, we focus on solutions that rely on general-purpose, *application agnostic*, trusted infrastructure. In particular, we can instantiate our solutions assuming the existence of a trusted *bulletin board*, which can be shared with a number of different applications, and which will provide a consistent (or *eventually* consistent) mapping between an epoch number i and a *commitment* d_i to the state of the registry at epoch i. Apart from this, our solutions will be *client auditable*, in that the client themselves verify global registry invariant proofs.

The challenges of IVC-based client auditability. A natural starting point to build client-auditable verifiable registries is to use *incrementally verifiable computation* (IVC) [Val08] via *recursive proofs* [BCCT13, BCTV14], following e.g. [CCDW20]. IVC enables the server to supply a commitment d_i to the state of the registry at epoch i, along with a succinct proof π_i that d_i represents a state which evolved from a genesis state d_0 through a sequence of transitions which preserve the registry's invariants. Clients can efficiently verify these invariant proofs on their own, without relying on dedicated third-party invariant auditors.

However, IVC proofs are, by themselves, not sufficient.Two users may come online at different epochs i and j and receive invariant proofs π_i and π_j, along with commitments d_i and d_j to different states of the registry. An IVC proof attests to invariant preservation for updates across *some* sequence of intermediate states leading to the states represented by d_i and d_j, respectively, but without additional verification, there is no guarantee that the intermediate states attested to in π_i and π_j are consistent with each other. To ensure that this is the case, a bulletin board could store the commitments $d_1, d_2, \ldots$, along with a *hash chain* $h_0, h_1, h_2, \ldots$, where $h_i = H(h_{i-1}, d_i)$ for some hash function H. Third-party auditors are responsible for verifying hash chain consistency, and the IVC proof π_i would attest that h_i commits to the unique hash sequence of valid registry states appearing on the bulletin board.

In practice, hash chain verification is only slightly more expensive than maintaining a bulletin board. A more important obstacle with IVC solutions is that generating invariant proofs is computa-

tionally expensive. Merkle trees are the predominant data structure for implementing an *authenticated dictionary* (AD) in existing verifiable registries [MBB+15, CDGM19, MKL+20]. Proving the invariant for a sequence of updates typically corresponds to verifying consistency of a sequence of Merkle paths. To achieve succinctness through IVC, the verification of Merkle paths is done within a succinct proof (in particular, a SNARK [Gro10, GGPR13]). However encoding the Merkle path verification into a circuit representation suitable for SNARKs results in a large circuit and concretely expensive proving times, ultimately translating to a verifiable registry with low update throughput (< 5 key updates/second). In contrast, the Certificate Transparency ecosystem requires throughput of approximately 60 key updates per second.

Our contributions. We aim to provide new verifiable registries which overcome the update throughput bottlenecks in IVC based solutions. Our new solutions will rely on the use of an RSA-based authenticated dictionary. Our main insight is a new cryptographic approach to produce succinct invariant proofs for large sequences of updates to an authenticated dictionary based on the KVaC key-value commitment construction from [AR20], opening up its use in the verifiable registry setting.

We then use our new insights to provide two systems, which we refer to as VeRSA-IVC and VeRSA-Amtz. In VeRSA-IVC, we show how KVaC and our new succinct proof can be combined with IVC to allow the server to produce succinct invariant proofs for client auditability at much higher throughput ($\sim$ 10-100$\times$ greater) than applying IVC to Merkle tree-based registries: the invariant proofs for the RSA dictionary encode as a constant-size circuit regardless of the number of updates, as opposed to a circuit linear in the number of updates for Merkle tree dictionaries (i.e., a Merkle path for each update), resulting in faster SNARK proving times.

Our second system, VeRSA-Amtz, provides instead a new amortized proving approach that dispenses with the need for IVC/SNARKs entirely, resulting in the first construction for efficient client auditability without IVC or generic SNARKs. We discuss in our related work section why prior solutions fall short of achieving this. Succinct invariant proofs for RSA authenticated dictionaries can be precomputed for carefully chosen sequences of updates over the lifetime of the registry in such a way that expensive computations for long sequences do not occur often, and any sequence of updates queried by a client can be served via a small number of precomputed invariant proofs for contiguous sequences. This alternate non-IVC approach enables even higher throughput in some

deployment contexts.

A novel challenge with VeRSA-Amtz is ensuring view consistency, as recursive SNARKs inherently gave us an easy solution via the use of hash chains. To this end, we introduce a new model of client-based auditing based on *checkpointing*. When a client comes online, they select a short (sublinear) sequence of checkpoint states between the current state and the state from when they were last online. The client can obtain a consistent view of the checkpoint digests thanks to the bulletin board, and then requests and verifies succinct proofs that the registry invariant is preserved between this sequence of checkpoints. Any two clients that individually perform these audits (over different checkpoint sequences) are guaranteed to have a consistent view up to their latest shared checkpoint; checkpoints are chosen so that two clients are guaranteed to have a shared checkpoint that is not too far behind their latest time online.

Our new auditing model relaxes consistency guarantees from previous approaches by allowing clients to temporarily accept an inconsistent state: the inconsistency is detected when the shared checkpoint catches up. But on the other hand, it enables clients to maintain eventually consistent views without expensive linear work and without relying on recursive SNARKs.

While our new proof techniques for RSA authenticated dictionaries allow for constructing client-auditable verifiable registries at high update throughput, computing lookup proofs for individual key-value mappings is more costly, naively requiring work linear in the size of the registry. We provide some deployment optimizations that help alleviate these costs with batching and caching, but ultimately this limitation means our RSA-based verifiable registries are better suited to transparency applications that need only maintain mappings on the order of millions, rather than Merkle tree approaches which can easily provide lookup proofs for billions of mappings. Nevertheless, examples of such settings where our constructions are immediately applicable include binary transparency (as of Jan 2022, Google Play Store included 3.3 million apps and Apple App Store included 2.1 million apps whereas Ubuntu's main repository included 106 thousand packages) or smaller messaging services such as Signal (40 million users). We demonstrate how our systems scale with increased resources, but new techniques or improved scaling through specialized hardware [SHT21, ZWZ+21] will likely be needed to make client-auditable verifiable registries practical for larger applications like Certificate Transparency (340 million domains) or WhatsApp (2 billion users).

We will present our results as modularly as possible, following the roadmap illustrated in

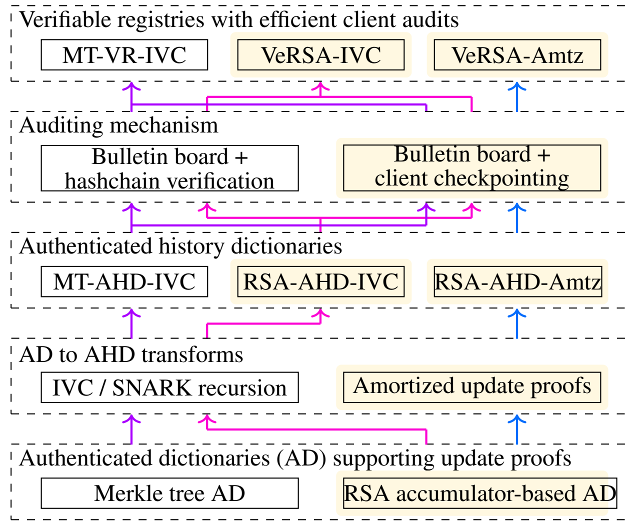

Figure 5.1: Overview of approaches to building verifiable registries efficiently auditable by clients. The highlighted boxes correspond to our new techniques and constructions, resulting in our two proposed verifiable registries, VeRSA-IVC and VeRSA-Amtz. The MT-VR-IVC verifiable registry can be considered as a baseline solution proposed in previous work [CCDW20]. We denote that the IVC-based registries can be instantiated via hashchain verification or via our new client checkpoint auditing mechanism.

Figure 5.1. In particular, we will start with the abstraction of an *authenticated dictionary* (AD) with an efficient invariant update proof, for which we provide an RSA instantiation by combining KVaC with our new update proofs. Then, we will show how to *generically* enhance such an AD into an *authenticated history dictionary* (AHD) which additionally allows for invariant proofs over the history of the dictionary, either via IVC or via our new amortization technique. Finally, we will combine the resulting AHDs with different trusted auditing mechanism (a plain bulletin board or one additionally verifying hash chains) to obtain our final systems.

Related work.

Registries from Merkle trees. Most previous proposals for verifiable registries (under various names) are constructed via Merkle trees and require auditors to do work linear in the total number of updates to the registry per epoch (at least one Merkle path verification per update) [BCK+14, KHP+13, Lau14, Rya14, CDGM19, MBB+15, MKL+20]. An exception is Merkle^2 [HHK+21] which reduces the per-epoch work of auditors to be logarithmic in the number of key updates; auditors verify a single Merkle extension proof. Merkle^2 fundamentally relies on a stronger assumption called *signature chains* in which key updates must be signed by an authorization key not controlled by the server.

This security policy does not allow users to recover if the authorization key is lost or compromised and hence may not be suitable for some deployments. In fact, in typical end-user applications it is a *requirement* that the server can unilaterally change a user's public key – a property needed for users to recover access if they lose their current device (and private keys) [MBB$^+$15]. We note that in applications where this restricted key update policy is applicable, Merkle^2 can be adapted using our amortized proving transform along with checkpoint auditing to construct an extremely efficient registry supporting efficient client audits (given a bulletin board); the Merkle extension proofs provide succinct invariant proofs for AD updates.

Privacy of registry contents has also been considered in prior work. Techniques to keep lookup keys private using verifiable random functions and lookup values private using commitments [MBB$^+$15, EMBB17] can be adapted directly to all of our constructions. While we do not consider other privacy notions such as hiding total directory size and update patterns [CDGM19], RSA accumulators may be better suited to this task than Merkle trees [BCD$^+$17]; we leave further investigation to future work.

Registries from algebraic accumulators. There are a few proposals using non-Merkle-based ADs. [TBP$^+$19] and Aardvark [LGG$^+$22] use bilinear pairing-based accumulators: [TBP$^+$19] admits succinct invariant proofs (logarithmic in the number of updates) which makes it a candidate for our amortized proving transform, however it is concretely expensive, while Aardvark, like Merkle-based approaches, provides linear invariant proofs (Aardvark improves parallelism of updates). RSA accumulators have also been proposed to construct registries with constant-sized verification work per epoch [BBF19, TXN20]. [BBF19] is not concretely efficient, requiring dictionary values to be committed bit-by-bit. [TXN20] propose a construction similar to [AR20] (both building on the line of work of [CF13,LM19]) but with two downsides: (1) Updating the digest requires computing an update hint which is similar in complexity to lookup proofs, and (2) the proposed invariant proof verifies that a dictionary contains a superset of keys of another dictionary, but does not verify properties about the mapped values of keys over time (a property necessary for our applications). In contrast, we build on the RSA AD of [AR20] which does not require update hints, and we propose invariant proofs for the versioned and append-only invariants allowing verifiable updates of a key's mapped value.

Regarding proving updates of values, [OWWB20] and [CFH$^+$22] provide techniques for proving

batch updates to an RSA accumulator with respect to a committed batch. [CFH$^+$22] improves over [OWWB20] by moving expensive linear-in-batch-size computation "out of" the generic SNARK. In our treatment of the verifiable registry setting, it is not necessary to prove that a specific set of keys were updated at each epoch (with respect to a committed batch of keys), rather only that *all* keys preserve the update invariant. Were this property desired, it may be possible to adapt these techniques to the authenticated dictionary primitive.

Applying SNARKs to registries. Verifiable computation [BFR$^+$13,LNS20,SAGL18] using SNARKs has also been proposed to lower per-epoch auditing costs by either (1) producing a succinct proof attesting to the updates for each epoch (so-called ZK rollups) [But, WGH$^+$] or (2) producing a recursive proof attesting to updates across all epochs committed in a hash chain [CCDW20, TKPS22]. These approaches require per-epoch auditors to perform only a SNARK verification or a simple hash verification, respectively. (Verdict [TKPS22] requires an inexpensive constraint accumulation check in addition to hash verification.) Swapping in our RSA AD (and invariant proof) over a Merkle-based AD would result in a smaller SNARK circuit encoding and more efficient proving for all of these approaches.

Finally, we note that while our focus has been on client-auditability, the succinct proofs provided by the above SNARK-based approaches or our new RSA AD approach may also be beneficial in making third-party auditing much more efficient. For example, per-epoch auditing may be inexpensive enough to run as a smart contract on a public blockchain. We leave a full evaluation of this setup to future work.

5.1 Auditing Public Key Infrastructure in Messaging

A *verifiable registry* [CDGM19, MKL$^+$20] maintains a collection D of key-value pairs (k, v) administered by a centralized[1] *server*. We assume that D contains at most one pair (k, v) for each k. The server periodically signs and publishes, at each *epoch*, a commitment (or *digest*) d_i to the registry state D_i on a public bulletin board (discussed shortly). Moving from epoch i to epoch $i+1$, means that one or more key-value pairs have been updated, i.e., (k, v) has been replaced by (k, v') or that an entry for a new k is added to D. There is an implicit notion that the updates and additions of these

[1] It would be possible to use a semi-centralized model in which a set of semi-trusted servers collaboratively maintain the registry using techniques from distributed consensus and threshold cryptography.

entries are the outcome of users requests—we do not specify these mechanisms further as they are application-specific. Also, we do not bound the number of updates of $D_{i+1} \setminus D_i$. Depending on the application context, a server may try to batch many updates into a single epoch, perhaps increasing epoch latency but achieving better throughput. Clients will then able to issue *lookup queries* to the registry and perform *monitoring* of entries to detect unexpected changes. We describe these below, after clarifying a few more high-level aspects of the model.

Threat model. Our primary goal is to guarantee a consistent view of the key-value mappings to all clients, and to allow for efficient monitoring of these mappings. The server is not trusted and may arbitrarily deviate from the protocol. Our goal is not to prevent attacks, in principle, but to ensure that they are *eventually* detected by some client accessing the system. This is particularly suitable for a *malicious-but-cautious* adversary [CDR14][2]. We do not attempt to guarantee availability, as a malicious server can simply refuse to respond to any queries. We also do not provide any privacy guarantees, though existing techniques for enhancing privacy can be implemented at the application-layer specification of (k, v) [MBB+15, EMBB17].

Bulletin board. As stated above, our solutions rely on a public *bulletin board* to prevent split-view attacks, in which a malicious server convinces user Alice to accept digest d_i and user Bob to accept digest $d_i' \neq d_i$ for the same epoch i. Both digests might be valid updates from a common ancestor d_j, but map a key to two distinct values. We assume that all digests $d_0, d_1, \ldots$ (i.e., one unique digest per epoch) are published by the server on the bulletin board, from which clients will read to maintain a consistent view, and that there exists an efficient mechanism for a client to read d_i for any i. Reliance on an out-of-band mechanism is necessary, in line with prior work on transparency systems [LLK13, MBB+15, CDGM19, MKL+20, LKMS04]. Bulletin boards, in particular, are a common assumption in cryptographic protocols [Ben87, CBM15, CGJ+17] which admits several possible implementations — e.g. a public blockchain [TD17] or a gossip protocol [STV+16, MKL+20]. The implementation of the bulletin board will require, either directly or indirectly, some trusted auditors ensuring that every epoch i is mapped to a unique d_i. In this work, all other auditing can be performed by clients themselves.

[2] A malicious-but-cautious adversary is willing to deviate from the protocol only in ways that will go undetected by user tests, e.g., if detection would lead to severe financial and/or reputational harm.

Basic lookups and monitoring. Clients can interact with the server to query a key[3] k at epoch i and retrieve the associated value v, along with a proof π of validity with respect to d_i and some additional metadata (such as a version number). Clients perform lookups at the current epoch i to learn the authoritative value for a given key. We envision particular applications where key-value entries are owned by some clients, e.g., if the registry implements a public key directory, a client will own the entry mapping their username to public key. We then assume clients continually look up their own keys to ensure that the mapped value is correct, a process called *monitoring*.

Associating certain invariant metadata (such as a version number) with each mapping enables efficient monitoring across digests even after the client has spent a long period offline, but requires that every digest preserves these invariants with respect to the prior digest. Past work has considered two such invariants. The *versioned* invariant [MBB+15,Bon16] associates with each key a version number that must be incremented whenever that key's value is updated. The *append-only* invariant [TBP+19, MKL+20] associates with each key an append-only list of the entire history of values for that key over the lifetime of the dictionary. Either invariant makes it easy to detect if a mapping has been modified; for example, in the versioned setting, if a client queries its own key at digest d_i and the associated metadata indicates the version number has not changed since the last digest d_j which the client queried, this guarantees the mapping has not changed during this period.In this work, we primarily focus on the simpler versioned invariant, observing that in most of our applications, it is sufficient to provide the most up-to-date value mapping.

Where monitoring can go wrong. It is instructive to consider concrete attacks a malicious server can mount to understand where monitoring can fail. The canonical attack we consider is sometimes called a *ghost value attack* (or ghost key attack) [MBB+15]. Consider a key owner that monitors their key at epochs i and j, and a second client that performs a lookup on the key at epoch ℓ where $i < \ell < j$. Suppose the key owner's expected mapped value for the key across epoch i to j is v. A ghost value attack occurs if the server can get the lookup client to accept a "ghost value" $v' \neq v$ for the key at epoch ℓ and then switch the value back to v at epoch j so that the owner's monitoring does not detect misbehavior. This attack is typically addressed, as mentioned above, through the use of invariant proofs that help with monitoring, e.g., detecting a change in version number. As long as (1)

[3]We use *key* to refer to the lookup key in a directory, e.g. a username. The *value* associated with that lookup key may itself be a cryptographic key in applications such as key transparency.

the view of epoch to digest mapping is consistent across clients and (2) the invariant is preserved between each digest, ghost value attacks will be detected. Thus, a ghost value attack can succeed if either of these assumptions fail – we next consider two attacks against these assumptions.

In a *split-view attack* [LKMS04], a server can publish different digests for an epoch to clients that are partitioned in different "worlds". In this attack, even if the invariant is preserved across the published digests in the key owner's world, it says nothing about the published digests in the lookup client's world, and monitoring will fail. We address the split-view attack by assuming a public bulletin board maintained by trusted auditors (see above) ensuring all clients have an eventually consistent view of the epoch-to-digest mapping — this appears to be a minimal assumption needed for a transparency system.

However, even with a consistent epoch-to-digest mapping, the question remains of who will verify invariant preservation between published digests. The server may mount an *oscillation attack* [MKL+20], in which it serves clients interleaving sequences of digests where each sequence preserves the invariant, but the two sequences interleaved do not preserve the invariant. For example, say the key owner is only served digests for even epochs, while the lookup client is served digests for odd epochs, and clients only verify the invariant holds for digests they are served. Monitoring will fail unless at some point an invariant proof is checked between an odd and even epoch digest. (Oscillation is of particular concern with asynchronous clients that come online at different times.) Prior work has addressed this by verifying invariant preservation between *every* consecutive pair of published digests using one of the following two approaches. The first approach simply assumes a set of trusted auditors that perform this task — we specify the use of outsourced trusted auditors because, typically, the invariant verification work (linear in the number of epochs) is considered too costly for the client to perform. The second approach, proposed in concurrent work [CCDW20, TKPS22], uses IVC with recursive SNARKs to allow for more efficient client verification. Specifically, registry digests are tied into a hash chain where $h_i = H(h_{i-1}, d_{i-1})$, and the pair (h_i, d_i) is stored for epoch i on the bulletin board. A succinct proof is created that attests to (1) invariant preservation between d_{i-1} and d_i, (2) inclusion in the hash chain $h_i = H(h_{i-1}, d_{i-1})$, and (3) recursive verification of the same proof for (h_{i-1}, d_{i-1}). By collision-resistance of the hash function, such a proof indirectly attests to the existence of a unique sequence of digests that each consecutively preserve the invariant. Even so, there is no guarantee that the sequence of digests attested to in the proof match the sequence

of digests published on the bulletin board. To prevent oscillation attacks, a client must additionally verify the hash chain posted on the bulletin board: if the hash chain is valid, then it must be that the sequence of published digests preserve the invariant. Verification of the hash chain is still linear in the number of epochs, but it is concretely inexpensive, and it is plausible a client may perform this task or that it may be outsourced to the trusted auditors maintaining the public bulletin board (e.g., via a smart contract).

Here, we put forward a novel approach to client-efficient auditing of invariant proofs to prevent oscillation attacks, which we overview next. Our approach assumes only a bulletin board (without relying on a hash chain), and will enable SNARK-free solutions such as VeRSA-Amtz.

Client checkpoint auditing. We introduce a new *checkpointing* technique, which we describe in detail in Section 5.4. Consider a client that was last online at epoch i and comes back online at epoch j. Instead of requiring the client to verify the invariant for all consecutive epochs in the range from i to j, the client will audit the invariant for a logarithmic number of *checkpoint digests* corresponding to certain canonical epochs between i and j. Crucially, these checkpoints are chosen so that any two overlapping ranges will share at least one checkpoint. This implicitly guarantees that, for any two clients, the invariant is preserved through the sequence of digests in their interleaved view up to their latest common checkpoint, and any oscillation that may have occurred since then will eventually be detected on future audits. We note that clients *may* temporarily accept two digests which do not preserve the invariant with respect to each other. Crucially, however, such an oscillation attack is *guaranteed* to eventually be detected at the next shared checkpoint.

5.2 Versioned Invariant Proofs for RSA Authenticated Dictionaries

We begin by constructing the first RSA-based authenticated dictionary that efficiently supports succinct versioned invariant proofs. Our starting point is the KVaC authenticated dictionary construction of Agrawal and Raghuraman [AR20]. We extend the original construction in two ways in order to make it suitable for use with verifiable registries. First, we show how to support efficient updates for a batch of key-value mappings ($([k_j, v_j]_j)$), instead of only a sole key-value update. Second, as our most significant contribution, we construct a succinct proof that a batch of updates applied to the dictionary preserve the versioned invariant. Building this proof, enables KVaC to achieve the

strong key binding security property needed for verifiable registries, in which key binding holds for adversarially chosen digests. Prior to this work, the construction was only secure with respect to *weak* key binding, i.e., digests that were produced honestly, limiting its applicability significantly.

In KVaC, key-value pairs are committed to with the following digest, where u represents a version number for the key, H is a collision-resistant hash function mapping keys to primes, and g is a member of an RSA quotient group:

$$d \leftarrow \left(g^{\left(\prod_i \mathsf{H}(k_i)^{u_i}\right) \cdot \left(\sum_i v_i / \mathsf{H}(k_i)\right)}, g^{\prod_i \mathsf{H}(k_i)^{u_i}} \right)$$

To update a key's value from v to $v+\delta$, the new digest $d' = (d_1^{\mathsf{H}(k)} d_2^{\delta}, d_2^{\mathsf{H}(k)})$ is computed, where the previous digest $d = (d_1, d_2)$.

Batching updates. When updating the values associated with many keys, we observe that instead of applying each update in sequence, all updates $[k, \delta]_i$ can be applied at once by the following:

$$Z \leftarrow \prod_i \mathsf{H}(k_i) \qquad \Delta \leftarrow \left(\prod_i \mathsf{H}(k_i)\right) \cdot \left(\sum_i \delta_i / \mathsf{H}(k_i)\right).$$

Then the batched update follows the same form as before, $d' = (d_1^Z d_2^{\Delta}, d_2^Z)$. We will take advantage of this form to construct succinct proofs for the versioned invariant.

Proving the versioned invariant. Informally, the versioned invariant enforces over an update that the only way to change a key's value is by increasing its version number. More formally, we define the invariant as follows with two constraints: (1) a key's version number does not decrease in an updated digest, and (2) two different values for a key cannot be shown for the same version number,

$$\Phi_{\mathsf{vsn}}(k, (v,u), (v',u')) = u < u' \lor (u = u' \land v = v'). \tag{5.1}$$

One approach to prove this invariant (and bootstrap strong key binding from weak key binding) is to prove that d' is the result of correctly applying the batch update procedure to d, i.e., that the update equations above hold, however it turns out that proving a weaker statement suffices. The prover constructs a proof of knowledge for the following relation between $d = (X_1, X_2)$ and updated digest $d' = (Y_1, Y_2)$:

$$\mathsf{R}_{\mathsf{KVaC}} = \left\{ ((X_1, X_2, Y_1, Y_2); (\alpha, \beta)) : Y_1 = X_1^{\alpha} X_2^{\beta} \land Y_2 = X_2^{\alpha} \right\}.$$

We show that it is computationally infeasible to produce a valid proof for this relation if the versioned

invariant is violated. This is a somewhat surprising result, as we do not enforce any extra structure on α and β, such as matching the structure of (Z, Δ) (which would result in a much more costly proof). Rather, simply proving knowledge of *any* α and β ensures that either the underlying pair of dictionary states do not violate the versioned invariant *or* that the prover has solved a computational problem related to factoring, breaking the Strong-RSA assumption.

We use the generalized knowledge of integer discrete log proof system from [BBF19] as the non-interactive proof of knowledge for $\mathsf{R}_{\mathsf{KVaC}}$. Importantly, this proof system, which leverages the algebraic structure of the RSA group, has a constant-time verification algorithm and constant-sized proof. This is a significant improvement over other Merkle-based [MBB+15, MKL+20] and bilinear pairing-based [TBP+19, LGG+22] constructions of authenticated dictionaries with versioned proofs.

Computing lookup proofs. Unfortunately, computing membership and non-membership proofs for keys from scratch is expensive – on the order of the combined number of keys with non-null values and number of past updates to the dictionary. Given a (non-)membership proof for a previous epoch, the proof can be updated to be valid for the current epoch in time linear in the number of key updates that have since occurred. However, even these updates can be expensive for the provider if many epochs have passed since a key's last query date. In our evaluation (Section 5.5), we show that for dictionaries with millions of keys, lookup proof computation costs are manageable; we discuss batch computation techniques that help alleviate these costs in the full version [TFZ+21].

Extending to the append-only invariant. While in this work, we focus on the versioned invariant, some applications may require the stronger append-only invariant that tracks the entire history of mapped values of a key. In the full version [TFZ+21], we propose an extension of KVaC for which we construct succinct append-only invariant proofs.

5.2.1 RSA Authenticated Dictionary

We make use of the key-value commitment KVaC from [AR20]; the construction pseudocode is given in Figure 5.2. The hash function H maps keys to primes of size 2^λ that are larger than the group order upper bound b. The space of values that can be committed to is the set of positive integers bounded above by b. [AR20] prove KVaC secure with respect to a weak key binding property in which the

$\mathsf{KVaC.Setup}(\lambda)$

$(a,b,\mathbb{G}) \leftarrow\!\!\$\; \mathsf{GGen}(\lambda)$
$g \leftarrow\!\!\$\; \mathbb{G}$
Return $(a,b,\mathbb{G},g)$

$\mathsf{KVaC.Init}()$

Return $(1,g)$

$\mathsf{KVaC.Comm}([(k,v,u)]_i)$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$C_1 \leftarrow g^{\sum_j \left(v_j z_j^{u_j-1} \prod_{i\neq j} z_i^{u_i}\right)}$
$C_2 \leftarrow g^{\prod_i z_i^{u_i}}$
Return (C_1,C_2)

$\mathsf{KVaC.ProveMem}([(k,v,u)]_i, m)$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$\pi_1 \leftarrow g^{\sum_{j\neq m} \left(v_j z_j^{u_j-1} \prod_{i\neq j,m} z_i^{u_i}\right)}$
$\pi_2 \leftarrow g^{\prod_{i\neq m} z_i^{u_i}}$
$(a,b) \leftarrow \mathsf{EEA}(\prod_{i\neq m} z_i^{u_i}, z_m)$
$\pi \leftarrow ((\pi_1,\pi_2),(g^b,a),u_m)$
Return π

$\mathsf{KVaC.VerifyMem}(C,(k,v),\pi)$

$z \leftarrow \mathsf{H}(k)$
$((\pi_1,\pi_2),(B,a),u) \leftarrow \pi$
$(C_1,C_2) \leftarrow C$
Return $\bigwedge \begin{pmatrix} (\pi_1)^{z^u}(\pi_2)^{v\cdot z^{u-1}} = C_1 \\ (\pi_2)^{z^u} = C_2 \\ (\pi_2)^a B^z = g \end{pmatrix}$

$\mathsf{KVaC.Upd}(C,(k,\delta))$

$z \leftarrow \mathsf{H}(k)$
$(C_1,C_2) \leftarrow C$
$C' \leftarrow (C_1^z C_2^\delta, C_2^z)$
Return C'

$\mathsf{KVaC.UpdateMemProof}((k,\pi),(k_\delta,\delta))$

$z \leftarrow \mathsf{H}(k)$
$((\pi_1,\pi_2),(B,a),u) \leftarrow \pi$
If $k = k_\delta$ then
  $\pi' \leftarrow ((\pi_1,\pi_2),(B,a),u+1)$
Else
  $z_\delta \leftarrow \mathsf{H}(k_\delta)$
  $(s,t) \leftarrow \mathsf{EEA}(z,z_\delta)$
  $q \leftarrow \lfloor \frac{at}{z} \rfloor$; $r \leftarrow at \mod z$
  $a' \leftarrow r$; $B' \leftarrow \pi_2^{as+qz_\delta} B$
  $\pi' \leftarrow ((\pi_1^{z_\delta}\pi_2^\delta, \pi_2^{z_\delta}),(B',a'),u)$
Return π'

$\mathsf{KVaC.ProveNonMem}([(k,v,u)]_i, k')$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$; $z' \leftarrow \mathsf{H}(k')$
$(a,b) \leftarrow \mathsf{EEA}(\prod_i z_i^{u_i}, z')$
Return (a, g^b)

$\mathsf{KVaC.VerNonMem}(C,k',\pi)$

$(a,B) \leftarrow \pi$
$z' \leftarrow \mathsf{H}(k')$; $(C_1,C_2) \leftarrow C$
Return $C_2^a B^{z'} = g$

$\mathsf{KVaC.UpdNonMemProof}((k',\pi),(k_\delta,\delta))$

$(a,B) \leftarrow \pi$
$z' \leftarrow \mathsf{H}(k')$; $z_\delta \leftarrow \mathsf{H}(k_\delta)$
$(s,t) \leftarrow \mathsf{EEA}(z',z_\delta)$
$q \leftarrow \lfloor \frac{at}{z} \rfloor$; $r \leftarrow at \mod z$
$a' \leftarrow r$; $B' \leftarrow \pi_2^{as+qz_\delta} B$
Return (a',B')

Figure 5.2: KVaC construction from [AR20]. The AD Lkup (resp. VerLkup) algorithm combines the prove (resp. verify) membership and non-membership algorithms.

$\mathsf{KVaC.BatchUpdate}(C,[(k,\delta)]_i))$

$[z]_i \leftarrow [\mathsf{H}(k)]_i$
$(C_1,C_2) \leftarrow C$
$Z \leftarrow \prod_i z_i$
$\Delta \leftarrow \sum_j \left(\delta_j \prod_{i\neq j} z_i\right)$
$C' \leftarrow (C_1^Z C_2^\Delta, C_2^Z)$
Return C'

$\mathsf{KVaC.UpdateMemProof}((k,\pi),(Z,\Delta))$

$z \leftarrow \mathsf{H}(k)$
$((\pi_1,\pi_2),(B,a),u) \leftarrow \pi$
$(s,t) \leftarrow \mathsf{EEA}(z,Z)$
$q \leftarrow \lfloor \frac{at}{z} \rfloor$; $r \leftarrow at \mod z$
$a' \leftarrow r$; $B' \leftarrow \pi_2^{as+qZ} B$
$\pi' \leftarrow ((\pi_1^Z \pi_2^\Delta, \pi_2^Z),(B',a'),u)$
Return π'

$\mathsf{KVaC.ProveUpdate}(C,C',(Z,\Delta))$

$(C_1,C_2) \leftarrow C$; $(C_1',C_2') \leftarrow C'$
$\pi \leftarrow \mathsf{BBF.Prove}((Z,\Delta),(C_1,C_2,C_1',C_2'))$
Return π

$\mathsf{KVaC.VerUpdate}(C,C',\pi)$

$(C_1,C_2) \leftarrow C$; $(C_1',C_2') \leftarrow C'$
Return $\mathsf{BBF.Ver}((C_1,C_2,C_1',C_2'),\pi)$

$\mathcal{R}_{\mathsf{KVaC}} = \left\{((X_1,X_2,Y_1,Y_2);(\alpha,\beta)) : Y_1 = X_1^\alpha X_2^\beta \wedge Y_2 = X_2^\alpha\right\}$

$\mathsf{BBF.Prove}((\alpha,\beta),(X_1,X_2,Y_1,Y_2))$

$s_a \leftarrow g^\alpha$; $s_b \leftarrow g^\beta$
$\ell \leftarrow \mathsf{H}_{\mathsf{Primes}}(X_1 \,\|\, X_2 \,\|\, Y_1 \,\|\, Y_2 \,\|\, s_a \,\|\, s_b)$
$q_a \leftarrow \lfloor \alpha/\ell \rfloor$; $r_a \leftarrow \alpha \mod \ell$
$q_b \leftarrow \lfloor \beta/\ell \rfloor$; $r_b \leftarrow \beta \mod \ell$
$W_a \leftarrow g^{q_a}$; $W_b \leftarrow g^{q_b}$
$W_1 \leftarrow X_1^{q_a} X_2^{q_b}$; $W_2 \leftarrow X_2^{q_a}$
$\pi \leftarrow (W_a,W_b,W_1,W_2,r_a,r_b,\ell)$
Return π

$\mathsf{BBF.Ver}((X_1,X_2,Y_1,Y_2),\pi)$

$\pi \leftarrow (W_a,W_b,W_1,W_2,r_a,r_b,\ell)$
$s_a \leftarrow W_a^\ell g^{r_a}$; $s_b \leftarrow W_b^\ell g^{r_b}$
Return $\bigwedge \begin{pmatrix} \ell = \mathsf{H}_{\mathsf{Primes}}(X_1 \,\|\, X_2 \,\|\, Y_1 \,\|\, Y_2 \,\|\, s_a \,\|\, s_b) \\ Y_1 = W_1^\ell X_1^{r_a} X_2^{r_b} \\ Y_2 = W_2^\ell X_2^{r_a} \end{pmatrix}$

Figure 5.3: Extension for KVaC to batch many key updates together (left). Extension to prove that key updates satisfy a versioned invariant (center) using the generalized proof of linear homomorphism from [BBF19], shown for the particular update homomorphism relevant to KVaC (right).

commitment must have been produced correctly, rather than adversarially. This is not sufficient for the verifiable registry setting; in the next section we show how to augment KVaC with update proofs to protect against adversarially generated commitments.

5.2.2 Versioned Invariant Update Proofs and Strong Key Binding

Figure 5.3 shows our protocol for proving updates preserve a versioned invariant. We use the generalized proof of linear homomorphism [BBF19] to prove that the commitment is updated only by a particular homomorphism that we show guarantees a versioned invariant. The proof of knowledge from [BBF19] is sound in the hidden order generic group model. We also show (in Figure 5.3) how to batch many key-value updates together such that the batched update follows the same homomorphic form as a single update. Individual membership proofs can be updated with respect to batched changes.

Next, we prove when KVaC construction from Figure 5.2 is combined with the update proofs from Figure 5.3, the construction achieves strong key binding and the versioned invariant is preserved. More specifically, to achieve strong key binding, we require that a digest from KVaC is accompanied with an update proof proving a valid update from the initial digest output from Init.

First, we will prove some useful lemmas.

Lemma 1. *[Shamir's trick] For any integer modulo N, given integers $u, v \in \mathbb{Z}_N^\times$ and $x, y \in \mathbb{Z}$, such that $u^x = v^y \mod N$ and $\gcd(x,y) = 1$, it is efficient to compute $w \in \mathbb{Z}_N^\times$ where $w^a = v \mod N$.*

Proof. Since $\gcd(x,y) = 1$, we can compute the Bézout coefficients $(a,b) \leftarrow \mathsf{EEA}(x,y)$ where $ax + by = 1$. Let $w = u^b v^a \mod n$, then

$$w^x = u^{bx} v^{ax} = (u^x)^b v^{ax} = (v^y)^b v^{ax} = v \pmod{N}.$$

□

Lemma 2. *[Non-trivial root of unity] For RSA quotient group $\mathbb{G}$ with elements of unknown order bounded above by b, given integers $u, v \in \mathbb{G}$ and prime $z > b$, if $u^z = v^z$, then $u = v$.*

Proof. Let $\alpha = u/v \in \mathbb{G}$. Then $\alpha^z = 1$. Since z is prime, if $\alpha \neq 1$, then z must be the order of α in $\mathbb{G}$. However, $z > b$, an upper bound on the order of elements in $\mathbb{G}$, which is not possible, so $\alpha = 1$ and $u = v$. □

Lemma 3. *[Coprime] For RSA quotient group $\mathbb{G}$, given integers $u, w \in \mathbb{G}$, random integer $v \in \mathbb{G}$, integers $a, b, c \in \mathbb{Z}$, and prime z, then if $u^{z^c} = v^a$ and $u^b w^z = v$, then $z^c \mid a$ and if let $d = a/z^c \in \mathbb{Z}$, then $u = v^d$ and $\gcd(z, d) = 1$.*

Proof. First, we prove that d exists, i.e., that $z^c \mid a$. Consider $(u^{z^{c-1}})^z = v^a$. If $z \nmid a$, then $\gcd(z, a) = 1$ and by Lemma 1, we can compute $x^z = v$ which wins the strong RSA security game. Therefore $z \mid a$ and $u^{z^{c-1}} = g^{a/z}$ by Lemma 2. We can repeat this argument for $(u^{z^{c-i}})^z = v^{a/z^{i-1}}$ for $i \in [2, c]$, ultimately arriving at $z^c \mid a$ and $u = v^{a/z^c} = v^d$.

Next, we show that $z \nmid d$. Consider $u^b w^z = v$ rewritten as $v^{bd-1} = w^{-z}$. If $z \mid d$, then $\gcd(bd - 1, -z) = 1$, and by Lemma 1, we can compute $x^z = v$ which again wins the strong RSA security game. Therefore, $z \nmid d$ meaning $\gcd(z, d) = 1$. □

Theorem 12. *For any adversary $\mathcal{A}$ against the versioned invariant soundness of* KVaC *augmented with proof of update from initialization, we give adversaries $\mathcal{B}$ and $\mathcal{C}$ such that*

$$\mathbf{Adv}^{inv}_{\mathsf{KVaC},\Phi_{\mathsf{vsn}},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\text{strong-rsa}}_{\mathsf{GGen},\mathcal{B}}(\lambda) + \mathbf{Adv}^{\text{sound}}_{\mathsf{BBF},\mathcal{C},\mathsf{X}}(\lambda),$$

where GGen *is the group generation algorithm for the RSA quotient group used in* KVaC *and* X *is the knowledge extractor for* BBF *[BBF19].*

Proof. First, we extract the update structure of the digests returned by adversary $\mathcal{A}$. Using the extractor X for BBF, we extract the values (α_A, β_A) from the update proof of $d_1 = C_A$ from the initial digest $(1, g)$. This gives us:

$$C_A = \left(g^{\beta_A}, g^{\alpha_A}\right).$$

Next, we extract the update structure of each of the updates from d_1 to d_m from the update proofs $[\pi_{\Phi,j}]_j^{m-1}$. Denote these extracted values as $[(\alpha_j, \beta_j)]_j^{m-1}$. We observe that using these values, we can write $d_m = C_B$ where we can define α_B and β_B as follows, as a single update from C_A:

$$C_B = \left(C_{A,1}^{\alpha_B} C_{A,2}^{\beta_B}, C_{A,2}^{\alpha_B}\right), \qquad \alpha_B = \prod_j^{m-1} \alpha_j, \qquad \beta_A = \sum_j^{m-1} \left(\beta_j \prod_{i \neq j} \alpha_i\right)$$

If the extractor fails, we build adversary $\mathcal{C}$ against the soundness of BBF.

The proof proceeds by considering each of the two winning conditions and showing that, in each case, a winning adversary can break strong RSA.

(1) $u_A > u_B$

(2) $v_A \neq v_B \wedge u_A = u_B$

<u>*Case 1*: $u_A > u_B$</u>

From the verification equations of π_A, we have that:

$$\pi_{A,2}^{z^{u_A}} = C_{A,2} = g^{\alpha_A}, \qquad \pi_{A,2}^{a_A} B_A^z = g\,.$$

Thus, by Lemma 3, we know that $\pi_{A,2} = g^{\alpha_A / z^{u_A}}$. Similarly, from the verification equations of π_B, we have that:

$$\pi_{B,2}^{z^{u_B}} = C_{B,2} = g^{\alpha_A \alpha_B}, \qquad \pi_{B,2}^{a_B} B_B^z = g\,.$$

Again, by Lemma 3, we have that $\pi_{B,2} = g^{\alpha_A \alpha_B / z^{u_B}}$ and $\gcd(\alpha_A \alpha_B / z^{u_B}, z) = 1$. Since $u_A > u_B$, we can construct group element u as follows:

$$u = \pi_{A,2}^{\alpha_B \cdot z^{u_A - u_B - 1}} \qquad \text{and then,} \qquad u^z = (\pi_{A,2}^{\alpha_B \cdot z^{u_A - u_B - 1}})^z = ((g^{\alpha_A / z^{u_A}})^{\alpha_B \cdot z^{u_A - u_B - 1}})^z = g^{\alpha_A \alpha_B / z^{u_B}}\,.$$

Since $\gcd(\alpha_A \alpha_B / z^{u_B}, z) = 1$, we can compute w from Lemma 1, where $w^z = g$ which wins the strong RSA security game.

<u>*Case 2*: $v_A \neq v_B \wedge u_A = u_B$</u>

Let $u = u_A = u_B$. By the verification equation of π_B, we have:

$$C_{B,1} = \pi_{B,1}^{z^u} \pi_{B,2}^{v_B z^{u-1}}$$

We also have, from the update proof and verification equations of π_A, that:

$$\begin{aligned} C_{B,1} &= C_{A,1}^{\alpha_B} C_{A,2}^{\beta_B} \\ &= \left(\pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\alpha_B v_A z^{u-1}}\right)\left(\pi_{A,2}^{\beta_B z^u}\right) \end{aligned}$$

We also can derive the following relation:

$$\pi_{A,2}^{z^u} = C_{A,2} \qquad \text{(by verification of } \pi_A\text{)}$$

$$\pi_{A,2}^{\alpha_B z^u} = C_{A,2}^{\alpha_B} = C_{B,2}$$

$$\pi_{B,2}^{z^u} = C_{B,2} \quad \text{(by verification of } \pi_B\text{)}$$

$$\pi_{A,2}^{\alpha_B} = \pi_{B,2} \quad \text{(by repeated application of Lemma 2)}$$

Putting this together we have as follows:

$$\begin{aligned}
\pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\alpha_B v_A z^{u-1}} \pi_{A,2}^{\beta_B z^u} &= \pi_{B,1}^{z^u} \pi_{B,2}^{v_B z^{u-1}} && \text{(by equality to } C_{B,1}\text{)} \\
\frac{\pi_{A,1}^{\alpha_B z^u} \pi_{A,2}^{\beta_B z^u}}{\pi_{B,1}^{z^u}} &= \frac{\pi_{B,2}^{v_B z^{u-1}}}{\pi_{A,2}^{\alpha_B v_A z^{u-1}}} \\
\frac{\pi_{A,1}^{\alpha_B z^u} B_A^{\beta_B z^u}}{\pi_{B,1}^{z^u}} &= \pi_{B,2}^{(v_B - v_A) z^{u-1}} && \text{(by relation between } \pi_{B,2} \text{ and } \pi_{A,2}\text{)} \\
\left(\left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z\right)^{z^{u-1}} &= \left(\pi_{B,2}^{v_B - v_A}\right)^{z^{u-1}} \\
\left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z &= \pi_{B,2}^{v_B - v_A} && \text{(by repeated application of Lemma 2)}
\end{aligned}$$

Thus, we have found a zth root of a non-trivial element. By Lemma 3, we have that $\pi_{B,2} = g^{\alpha_A \alpha_B / z^u}$ where $\gcd(\alpha_A \alpha_B / z^u, z) = 1$. This gives us

$$\left(\frac{\pi_{A,1}^{\alpha_B} \pi_{A,2}^{\beta_B}}{\pi_{B,1}}\right)^z = g^{\frac{(v_B - v_A)\alpha_A \alpha_B}{z^u}} .$$

Since z is prime and the domain of values is chosen to be smaller than all z, we also have that $\gcd(v_A - v_B, z) = 1$, and therefore by Lemma 1, we can compute w where $w^z = g$ winning the strong RSA security game.

□

Theorem 13. *For any adversary $\mathcal{A}$ against the strong key binding of* KVaC *augmented with proof of update from initialization, we give adversaries $\mathcal{B}$ and $\mathcal{C}$ such that*

$$\mathbf{Adv}^{\textit{bind}}_{\mathsf{KVaC},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\text{strong-rsa}}_{\mathsf{GGen},\mathcal{B}}(\lambda) + \mathbf{Adv}^{\text{sound}}_{\mathsf{BBF},\mathcal{C},\mathsf{X}}(\lambda),$$

where GGen *is the group generation algorithm for the RSA quotient group used in* KVaC *and* X *is the knowledge extractor for* BBF *[BBF19].*

Proof. The proof follows similarly to that of Theorem 12. Using extractor X for BBF, we extract values (α, β) from the update proof of d from the initial digest $(1, g)$, giving us: $d = C = (g^\beta, g^\alpha)$. We then proceed by considering the following two winning conditions; in each case, a winning adversary can break strong RSA.

(1) $u_A \neq u_B$

(2) $v_A \neq v_B \wedge u_A = u_B$

Case 1: $u_A \neq u_B$

The first case follows similarly to (Case 1) of Theorem 12. From the verification equations of π_A, we have that:

$$\pi_{A,2}^{z^{u_A}} = C_2 = g^\alpha, \qquad \pi_{A,2}^{a_A} B_A^z = g\,.$$

Thus, by Lemma 3, we know that $z^{u_A} \mid \alpha$. Similarly, from the verification equations of π_B, we have that:

$$\pi_{B,2}^{z^{u_B}} = C_2 = g^\alpha, \qquad \pi_{B,2}^{a_B} B_B^z = g\,.$$

Again, by Lemma 3, we have that $\gcd(\alpha/z^{u_B}, z) = 1$. However, this is a contradiction. Wlog say $u_A > u_B$, then since $z^{u_A} \mid \alpha$, it cannot be that $\gcd(\alpha/z^{u_B}, z) = 1$.

Case 2: $v_A \neq v_B \wedge u_A = u_B$

The second case follows exactly from (Case 2) of Theorem 12 where $\alpha_A = \alpha$, $\beta_A = \beta$, $\alpha_B = 1$, and $\beta_B = 0$. □

5.3 Authenticated History Dictionaries

In this section we will define an *authenticated history dictionary* (AHD), the novel cryptographic primitive behind our verifiable registry system, and present several constructions of this primitive from authenticated dictionaries.

5.3.1 Syntax and Security Notions

An AHD commits not only to its current state, but also to all previous states in its history. It is also able to efficiently provide update invariant proofs between any sequence of previous states. As for

authenticated dictionaries, we define an invariant Φ as a boolean function on input k, v_i, v_j that outputs 1 if the invariant is preserved; we require the invariant to be preserved for all keys. Again, in this work, we will be interested in the versioned invariant Φ_{vsn} (Equation 5.1). An AHD is defined by the following set of algorithms:

- $pp \leftarrow_{\$} \mathsf{Setup}(\lambda)$: The setup algorithm takes a security parameter and returns public parameters.
- $(d_0, st_0) \leftarrow \mathsf{Init}()$: The initialization algorithm returns an initial digest to the empty dictionary.
- $(d_{i+1}, st_{i+1}) \leftarrow \mathsf{Upd}([k_j, v_j]_j : st_i)$: The update algorithm updates the dictionary values for input keys $\{k_j\}$ to the values $\{v_j\}$ and outputs a new digest d_{i+1} representing the new dictionary history for epoch $i+1$.
- $(v, \pi_{\mathsf{lkup}}) \leftarrow \mathsf{Lkup}(k : st_i)$: The lookup algorithm returns the value v associated with k along with a membership proof π_{lkup}. If the k is not present in the dictionary, v is set to $\perp$ and a non-membership proof is provided.
- $0/1 \leftarrow \mathsf{VerLkup}(d_i, k, v, \pi_{\mathsf{lkup}})$: The lookup verification algorithm verifies the key-value mapping in d_i.
- $\pi_{\mathsf{hist}} \leftarrow \mathsf{ProveHist}([c_j]_j^m : st_i)$: The prove history algorithm takes as input an ordered list of checkpoint epochs, $c_1 < \ldots < c_m < i$, and provides a proof that the digest at each checkpoint is included in the committed history.
- $0/1 \leftarrow \mathsf{VerHist}(d_i, [(c_j, d_{c_j})]_j^m, \pi_{\mathsf{hist}})$: The history verification algorithm verifies the ordered list of checkpoint digests are included in the history of digest d_i.
- $\pi_\Phi \leftarrow \mathsf{ProveInv}([c_j]_j^m : st_i)$: The prove invariant algorithm takes as input an ordered list of checkpoint epochs, $c_1 < \ldots < c_m \leq i$, and provides a proof that the invariant Φ is preserved between the dictionary states of each pair of digests in sequence: $(d_{c_j}, d_{c_{j+1}})$.
- $0/1 \leftarrow \mathsf{VerInv}(d_i, [(c_j, d_{c_j})]_j^m, \pi_\Phi)$: The invariant verification algorithm verifies the invariant is preserved between the sequence of ordered checkpoint digests.

An important feature of the AHD syntax and semantics is allowing querying of history and invariant properties for previous states. While critical to support client auditing as clients often come online after long periods of disconnection, this functionality is what creates the main challenge in coming up with efficient constructions.

In terms of correctness, informally, the dictionary should correctly update its key-value mappings

Game $\text{BIND}^{\mathcal{A}}_{\mathsf{AHD}}(\lambda)$

$pp \leftarrow\$ \mathsf{AHD.Setup}(\lambda)$

$(k, d, (v_A, \pi_A), (v_B, \pi_B)) \leftarrow\$ \mathcal{A}(pp)$

Return

$$\bigwedge \begin{pmatrix} \mathsf{AHD.VerLkup}(d_A, k, v_A, \pi_A) \\ \mathsf{AHD.VerLkup}(d_B, k, v_B, \pi_B) \\ v_A \neq v_B \end{pmatrix}$$

Game $\text{HISTBIND}^{\mathcal{A}}_{\mathsf{AHD}}(\lambda)$

$pp \leftarrow\$ \mathsf{AHD.Setup}(\lambda)$; $\mathsf{win} \leftarrow 0$

$(d, st) \leftarrow\$ \mathcal{A}_1(pp)$

$\mathcal{D} \leftarrow \mathcal{D} \cup [d]$

$\mathcal{A}_2^{\text{PROVEHIST}}(st)$

Return win

Oracle $\text{PROVEHIST}(d', [(c_j, d_j)]_j^m, \pi)$

Require $d' \in \mathcal{D}$

If $\mathsf{AHD.VerHist}(d', [(c_j, d_j)]_j^m, \pi)$ then

- For all $j \in [1, m]$:
  - If $c_j \in V$ and $V[c_j] \neq d_j$ then $\mathsf{win} \leftarrow 1$
  - $V[c_j] \leftarrow d_j$
  - $\mathcal{D} \leftarrow \mathcal{D} \cup [d_j]$

Game $\text{INVSOUND}^{\mathcal{A}}_{\mathsf{AHD},\Phi}(\lambda)$

$pp \leftarrow\$ \mathsf{AHD.Setup}(\lambda)$

$$\begin{pmatrix} k, d, \left[[(c_{i,j}, d_{i,j})]_j^{m_i}\right]_i^n, [\pi_{\Phi,i}, \pi_{\mathsf{hist},i}]_i^n \\ (i_A, j_A, v_A, \pi_A), (i_B, j_B, v_B, \pi_B) \end{pmatrix} \leftarrow\$ \mathcal{A}(pp)$$

$d_{n+1,1} \leftarrow d$

Return

$$\bigwedge \begin{pmatrix} \mathsf{AHD.VerLkup}(d_{i_A,j_A}, k, v_A, \pi_A) \\ \mathsf{AHD.VerLkup}(d_{i_B,j_B}, k, v_B, \pi_B) \\ \left[\mathsf{AHD.VerInv}(d_{i+1,1}, [(c_{i,j}, d_{i,j})]_j^{m_i}, \pi_{\Phi,i})\right]_i^n \\ \left[\mathsf{AHD.VerHist}(d_{i+1,1}, [(c_{i,j}, d_{i,j})]_j^{m_i}, \pi_{\mathsf{hist},i})\right]_i^n \\ \Phi(k, v_A, v_B) \neq 1 \\ i_A < i_B \vee (i_A = i_B \wedge j_A \leq j_B) \end{pmatrix}$$

Figure 5.4: Security games for strong key binding (left), history binding (middle), and invariant preservation (right) for authenticated history dictionaries.

and lookups should return the latest value added. Previous digests should be correctly committed to in the appropriate epoch position in history. And lastly, the proofs produced by the proving algorithms should pass their accompanying verification algorithms.

In terms of security, we define three properties. The first two properties are analogous to the security properties of ADs. First, an AHD must satisfy *key binding*, which is defined equivalently to as in ADs: it should not be possible to provide valid lookup proofs to two different values for a key in a digest. Second, *invariant soundness* requires that it should not be possible to produce a valid invariant proof for a sequence of checkpoints such that the invariant is not preserved between any two checkpoint digests. The last property is *history binding*, which requires that it should not be possible to provide two valid history proofs for a digest including a different checkpoint digest at the same checkpoint epoch.

We formally define these properties as pseudocode security games provided in Figure 5.4. The key binding game is equivalent to that of authenticated dictionaries. The history binding game tasks an adversary with producing two history proofs (or sequences of history proofs) that verify with two different digests for the same epoch. The invariant soundness game tasks an adversary with producing lookup proofs for a key with two values that do not satisfy the invariant, while also proving the invariant holds between the two epochs for which the lookup proofs were provided. We define an

adversary's advantage against these games, respectively, as:

$$\mathbf{Adv}^{\mathsf{bind}}_{\mathsf{AHD},\mathcal{A}}(\lambda) = \Pr[\textsc{Bind}^{\mathsf{AHD}}_{\mathcal{A}}(\lambda) = 1] \,, \qquad \mathbf{Adv}^{\mathsf{hbind}}_{\mathsf{AHD},\mathcal{A}}(\lambda) = \Pr[\textsc{HistBind}^{\mathsf{AHD}}_{\mathcal{A}}(\lambda) = 1] \,,$$

$$\mathbf{Adv}^{\mathsf{inv}}_{\mathsf{AHD},\Phi,\mathcal{A}}(\lambda) = \Pr[\textsc{InvSound}^{\mathsf{AHD},\Phi}_{\mathcal{A}}(\lambda) = 1] \,.$$

5.3.2 AHD Constructions

Towards a Generic Construction

We begin by discussing useful building blocks and strawman solutions for constructing an AHD from an underlying AD.

Composing an AD with a history commitment. A core additional functionality AHDs provide over ADs is the ability to track and commit to the history of previous states. As such, a natural starting point to build an AHD is to combine an AD with an append-only vector commitment (VC), committing the digest of the AD at time step i to the ith position of the vector commitment; we will refer to the vector commitment as a *history commitment*.

More specifically, consider an AHD made of an authenticated dictionary D and a vector commitment L: the digest of the AHD is a pair of digests (or hash of pair), one from an authenticated dictionary and the other of the history commitment: $(d_{\mathsf{AD}}, d_{\mathsf{VC}})$. To perform a set of key-value updates $[k_i, v_i]_i$, first, a new AD digest is computed by updating the AD, $(d'_{\mathsf{AD}}, D') \leftarrow \mathsf{AD.Upd}(D, [k_i, v_i]_i)$. Then, the vector commitment is updated to append the old digest, $(d'_{\mathsf{VC}}, L') \leftarrow \mathsf{VC.Upd}(L, [(d_{\mathsf{AD}}, d_{\mathsf{VC}})])$. The new AHD digest is set as $(d'_{\mathsf{AD}}, d'_{VC})$.

This construction also supports succinct proofs to $\mathsf{AHD.ProveHist}$ queries for arbitrary checkpoints. A prefix proof using $\mathsf{VC.ProveUpd}$ is computed for each checkpoint with respect to the current state. For the Merkle tree instantiation of VC, these proofs both can be computed and verified in time and are of size $\mathcal{O}(\log N)$ where N is length of the vector. This basic combination of AD and history commitment form the basis of our proposed constructions. The pseudocode details are given in Figure 5.5; and we provide proof sketches for history binding and key binding below.

Challenge of succinct invariant proofs. Unfortunately, it is not straightforward how to provide succinct invariant proofs for arbitrary checkpoints in response to $\mathsf{AHD.ProveInv}$. Recall, an AD can

be augmented to provide invariant proofs for updates. An invariant proof π_i can be computed during each epoch update for $d_{\mathsf{AD},i-1}$ to $d_{\mathsf{AD},i}$. For a queried pair of checkpoints (c_j, c_{j+1}), the sequence of epoch invariant proofs $[\pi_i]_{i=c_j}^{c_{j+1}}$ together attest to invariant preservation for d_{AD,c_j} to $d_{\mathsf{AD},c_{j+1}}$. However, this would not be succinct, ultimately leading to a proof of size linear in the range of epochs the checkpoints are over.

Alternatively, it is also not efficient to compute a fresh invariant proof for pairs of checkpoints (c_j, c_{j+1}) on the fly in response to a $\mathsf{ProveInv}$ query. Each invariant proof is computed in time linear in the number of key-value updates made to the dictionary.

Instead, we will need different approaches. We present two generic constructions for AHDs from ADs. First, we present a construction for succinct invariant proofs based on IVC. It is our most general solution and is compatible with any AD that supports the invariant proof. Second, we present a construction based on amortized proving of invariant preservation over power-of-two ranges of epochs: an invariant proof for any pair of checkpoints can be provided as a $\log N$ sequence of precomputed proofs. This approach dispenses with the heavyweight machinery of IVC, but requires that the underlying AD supports a *succint* invariant proof. This is the case for our new RSA construction (Section 5.2), however does not hold for Merkle tree ADs.

AHDs from SNARK Recursion

Our first construction is from IVC; for concreteness, we present the construction using recursive proofs [BCCT13, BCTV14], the prevailing approach to IVC. IVC allows constructing a succinct proof of an output (digest) that attests to its correct computation over a series of steps (invariant preserved over epochs). IVC has previously been proposed for producing succinct proofs for verifiers of invariant-based ledger systems [CCDW20], a more general case of verifiable registries.

The starting point of our construction $\mathsf{AHD}_{\mathsf{IVC}}$ is an AD with history commitment (described in the previous Section 5.3.2). On each epoch update, in addition to updating the digests as before, a recursive SNARK proof is computed attesting to invariant preservation. Namely, at epoch i, the proofs π_Φ from $\mathsf{AD.Upd}$ showing the updated key-values satisfy the invariant and π_{hist} from $\mathsf{VC.ProveUpd}$ showing the new AD digest was appended to history commitment are computed. Then using a SNARK, $\pi_{\mathsf{SNARK},i}$ proves that (1) π_Φ verifies with respect to $d_{\mathsf{AD},i-1}$ and $d_{\mathsf{AD},i}$, (2) π_{hist} verifies with respect to $d_{\mathsf{VC},i-1}$ and $d_{\mathsf{VC},i}$, and (3) that recursively verifies a SNARK $\pi_{\mathsf{SNARK},i-1}$ for d_{i-1}.

Informally, this SNARK proves "the invariant is preserved across the sequence of digests committed to in the history commitment". The complexity of the recursive relation is proportional to the combined complexity of the SNARK verification algorithm, vector commitment update verification, and importantly, the AD invariant verification algorithm, which differs significantly between a Merkle tree-based AD and our new RSA AD.

Completing the picture, the proof of invariant preservation over a sequence of checkpoints $[c_j]_j$ consists of two parts: (1) the most recent SNARK proved for the current epoch i, $\pi_{\mathsf{SNARK},i}$, and (2) a lookup proof in the history commitment for each of the checkpoints, proving the value at index c_j is d_{AD,c_j}. Intuitively, the SNARK proves that the invariant is preserved across digests in the history commitment, and the lookup proofs reveal the checkpoint digests are indeed included in the history. A protocol description for the $\mathsf{AHD}_{\mathsf{IVC}}$ construction is given in Figure 5.5.

AHDs from Amortized Proving

Recall the two strawman proving approaches for providing an invariant proof for checkpoints (c_j, c_{j+1}) from Section 5.3.2. The first was to provide a sequence of "epoch invariant proofs", one for each epoch between c_j and c_{j+1}: these can be efficiently precomputed, but do not result in a succinct proof. The second was to directly prove an invariant proof for the key-value updates in the range from c_j to c_{j+1}: this cannot be precomputed efficiently as there are quadratically many possible checkpoint ranges that could be queried, however would result in a succinct proof if the invariant proving algorithm of the underlying AD is succinct (as it is for our RSA AD from Section 5.2).

In this section, we propose a construction $\mathsf{AHD}_{\mathsf{Amtz}}$ that serves as a middle ground between these two approaches. Instead of attempting to precompute proofs for *all* possible start and end epoch ranges, only proofs for compact subranges will be precomputed. Recall a compact range for a range (c_j, c_{j+1}) produces a succinct sequence of subranges $[(L_\ell, R_\ell)]_\ell^m$ that "span" (c_j, c_{j+1}); that is, $L_1 = c_j$, $R_m = c_{j+1}$, and $R_i = L_{i+1}$ for all $1 \leq i < m < \log(c_{j+1} - c_j)$. Importantly, each compact subrange is guaranteed to be of the form: $(L_i = a_i \cdot 2^{b_i}, R_i = L_i + 2^{b_i})$ for non-negative integers (a_i, b_i).

Precomputing invariant proofs for just these compact subranges is amortized efficient. The structure of compact subranges – that they start on multiples of powers-of-two and are of length power-of-two – mean that there are only linear (in the number of epochs) such subranges. At epoch N,

Protocol: $\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD}, \mathsf{VC}, \mathsf{SNARK}]$

Setup: The public parameters of the scheme consist of the public parameters of its underlying components: $pp \leftarrow (pp_{\mathsf{AD}}, pp_{\mathsf{VC}}, (pk, vk)_{\mathsf{SNARK}})$.

Init: The dictionary is initialized with an empty authenticated dictionary and empty vector commitment, returning an initial digest $d_0 = (d_{\mathsf{AD},0}, d_{\mathsf{VC},0})$. It stores the following as its current state st_i:

- $st_{\mathsf{AD},i}$: state of the AD representing current state of key-value mapping.
- $st_{\mathsf{VC},i}$: state of the VC representing list of previous epoch digests.
- $\pi_{\mathsf{SNARK},i}$: SNARK proof attesting to invariant preservation for latest epoch.

$\mathsf{Upd}([k_j, v_j]_j : st_i)$:

(1) The AD is updated with the new key-value mappings: $(d_{\mathsf{AD},i+1}, \pi_\Phi, st_{\mathsf{AD},i+1}) \leftarrow \mathsf{AD.Upd}([k_j, v_j]_j : st_{\mathsf{AD},i})$.

(2) The previous digest is appended to the history commitment:
$(d_{\mathsf{VC},i+1}, st_{\mathsf{VC},i+1}) \leftarrow \mathsf{Upd}([d_i] : st_{\mathsf{VC},i})$, and $\pi_{\mathsf{hist}} \leftarrow \mathsf{ProveUpd}(i : st_{\mathsf{VC},i+1})$, $\pi_{\mathsf{lkup}} \leftarrow \mathsf{Lkup}(i : st_{\mathsf{VC},i+1})$.

(3) A new SNARK $\pi_{\mathsf{SNARK},i+1}$ is computed attesting to invariant preservation for new digest $d_{i+1} = (d_{\mathsf{AD},i+1}, d_{VC,i+1})$, proving the following relation:

$$\mathsf{R}_{\mathsf{SNARK}} = \left\{ \begin{array}{r} \Big(\;(d_{i+1}), (d_i, \pi_\Phi, \pi_{\mathsf{hist}}, \pi_{\mathsf{SNARK},i})\; \Big) : \\ \mathsf{AD.VerUpd}(d_{\mathsf{AD},i}, d_{\mathsf{AD},i+1}, \pi_\Phi) \\ \mathsf{VC.VerUpd}(d_{\mathsf{VC},i}, d_{\mathsf{VC},i+1}, i, \pi_{\mathsf{hist}}) \\ \mathsf{VC.VerLkup}(d_{\mathsf{VC},i+1}, i, d_i, \pi_{\mathsf{lkup}}) \\ \mathsf{SNARK.Ver}(vk_{\mathsf{SNARK}}, d_i, \pi_{\mathsf{SNARK},i}) \end{array} \right\} .$$

$\mathsf{ProveInv}([c_j]_j^m : st_i)$:

(1) For each checkpoint, a lookup proof in the history commitment for the checkpoint index is computed: $[\pi_{\mathsf{lkup},j} \leftarrow \mathsf{VC.Lkup}(c_j : st_{\mathsf{VC},i})]_j^m$.

(2) Proof $\pi_\Phi \leftarrow (\pi_{\mathsf{SNARK},i}, [\pi_{\mathsf{lkup},j}]_j^m)$ is returned.

$\mathsf{VerInv}(d_i, [(c_j, d_{c_j})]_j^m, \pi_\Phi = (\pi_{\mathsf{SNARK}}, [\pi_{\mathsf{lkup},j}]_j^m))$:

(1) The SNARK proof is verified: $\mathsf{SNARK.Ver}(vk_{\mathsf{SNARK}}, d_i, \pi_{\mathsf{SNARK}})$.

(2) The history commitment lookup proof for each checkpoint digest is verified: $[\mathsf{VC.VerLkup}(d_{\mathsf{VC},i}, c_j, d_{\mathsf{AD},c_j}, \pi_{\mathsf{lkup},j})]_j^m$.

$\mathsf{Lkup}(k : st_i)$ and $\mathsf{VerLkup}(d_i, k, v, \pi_{\mathsf{lkup}})$: Lookup and lookup verification use the lookup algorithms of the underlying AD over $st_{\mathsf{AD},i}$ and $d_{\mathsf{AD},i}$:

$$(v, \pi_{\mathsf{lkup}}) \leftarrow \mathsf{AD.Lkup}(k : st_{\mathsf{AD},i}), \qquad \mathsf{AD.VerLkup}(d_{\mathsf{AD},i}, k, v, \pi_{\mathsf{lkup}})$$

$\mathsf{ProveHist}([c_j]_j^m : st_i)$: For each checkpoint, an lookup proof for the history commitment is provided: $\pi_{\mathsf{hist}} = [(\pi_{\mathsf{lkup},j}, \pi_{\mathsf{hist},j})]_j^m$

$$\pi_{\mathsf{lkup},j} \leftarrow \mathsf{VC.Lkup}(c_j : st_{\mathsf{VC},i}), \;\; \pi_{\mathsf{hist},j} \leftarrow \mathsf{VC.ProveUpd}(c_j : st_{\mathsf{VC},i}) .$$

$\mathsf{VerHist}(d_i, [(c_j, d_{c_j})]_j^m, \pi_{\mathsf{hist}} = [(\pi_{\mathsf{lkup},j}, \pi_{\mathsf{hist},j})]_j^m)$:

$$[\mathsf{VC.VerLkup}(d_{\mathsf{VC},i}, c_j, d_{c_j}, \pi_{\mathsf{lkup},j}), \;\; \mathsf{VC.VerUpd}(d_{\mathsf{VC},i}, d_{c_j}, c_j, \pi_{\mathsf{hist},j})]_j^m$$

Figure 5.5: Generic construction of an AHD from an AD using incrementally-verifiable computation through recursive SNARKS. The history of the AHD is committed to using an append-only vector commitment referred to as a history commitment.

Protocol: $\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}]$

Setup: The public parameters of the scheme consist of the public parameters of its underlying components: $pp \leftarrow (pp_{\mathsf{AD}}, pp_{\mathsf{VC}})$.

Init: The dictionary is initialized with an empty authenticated dictionary and empty vector commitment, returning an initial digest $d_0 = (d_{\mathsf{AD},0}, d_{\mathsf{VC},0})$. It stores the following as its current state st_i:

- $L_{\mathsf{AD}} = [st_{\mathsf{AD},\ell}, d_{\mathsf{AD},\ell}]_\ell^i$: state of the AD at each epoch.
- $st_{\mathsf{VC},i}$: state of the VC representing list of previous epoch digests.
- $L_k = [[k_{\ell,j}, v_{\ell,j}]_j^{m_\ell}]_\ell^i$: list of key-value updates applied at each epoch.
- T_Φ: table of precomputed invariant proofs for all compact subranges.

$\mathsf{Upd}([k_j, v_j]_j : st_i)$:

(1) The AD is updated with the new key-value mappings:
$(d_{\mathsf{AD},i+1}, \pi_\Phi, st_{\mathsf{AD},i+1}) \leftarrow \mathsf{AD.Upd}([k_j, v_j]_j : st_{\mathsf{AD},i})$.

(2) The new AD digest is appended to the history commitment:
$(d_{\mathsf{VC},i+1}, st_{\mathsf{VC},i+1}) \leftarrow \mathsf{Upd}([d_{\mathsf{AD},i+1}] : st_{\mathsf{VC},i})$.

(3) Compute and store an invariant proof for the key updates applied during every compact subrange of epochs that $i+1$ closes, i.e., $[L_j]_j^m$ such that there exists (a_j, b_j) where $L_j = a_j \cdot 2^{b_j}$ and $L_j + 2^{b_j} = i+1$:

$$T_\Phi[L_j, i+1] \leftarrow \mathsf{AD.ProveUpd}([[k_{\ell,k}, v_{\ell,k}]_k]_{\ell=L_j}^{i+1}, st_{\mathsf{AD},L_j}).$$

(4) The new digest $d_{i+1} = (d_{\mathsf{AD},i+1}, d_{VC,i+1})$ is returned.

$\mathsf{ProveInv}([c_j]_j^m : st_i)$: For each checkpoint pair (c_j, c_{j+1}) for $1 \leq j < m$ compute $\pi_{\Phi,j}$ then return $\pi_\Phi \leftarrow [\pi_{\Phi,j}]_j^m$:

(1) Compute the n_j compact subranges that span $(c_j, c_{j+1}]$:

$$[(L_{j,\ell}, R_{j,\ell})]_\ell^{n_j} \leftarrow \mathsf{CompactR}((c_j, c_{j+1})).$$

(2) Construct an invariant proof for (c_j, c_{j+1}) with the precomputed invariant proofs of each compact subrange: $\pi_{\Phi,j} = [T_\Phi[L_{j,\ell}, R_{j,\ell}], d_{L_{j,\ell}}]_\ell^{n_j}$.

$\mathsf{VerInv}(d_i, [(c_j, d_{c_j})]_j^m, \pi_\Phi = \left[\left[(\pi_{\Phi,j,\ell}), d_{\mathsf{AD},j,\ell})\right]_\ell^{n_j}\right]_j^m)$: For each checkpoint pair (c_j, c_{j+1}) for $1 \leq j < m$:

(1) Verify compact range endpoints: $d_{c_j} = d_{\Phi,j,1}$ and $d_{c_{j+1}} = d_{\Phi,j,n_j}$.

(2) Verify each compact subrange invariant proof:

$$[\mathsf{AD.VerUpd}(d_{\Phi,j,\ell}, d_{\Phi,j,\ell+1}, \pi_{\Phi,j,\ell})]_\ell^{n_j-1}.$$

Figure 5.6: Generic construction of an AHD from an AD using amortized proving of invariant preservation over compact subranges. The underlying AD must support succinct invariant proofs (e.g., as in the new RSA AD construction).

there are $\leq N$ compact subranges, $\sum_{i=1}^{\lg N} N/2^i$, and the sum of their lengths is $\leq N \lg N$. Invariant proofs for ranges of length n are computed in work linear in n. Thus, by a classic amortization argument, for an AHD at epoch N, the total work to compute invariant proofs for all N compact subranges can be amortized efficiently to a cost of $O(\lg N)$ for each new published epoch [Ove83].

Given precomputed invariant proofs for compact subranges, a succinct invariant proof can be constructed for any pair of checkpoints (c_j, c_{j+1}) simply by providing the precomputed invariant

proofs for each compact subrange in compact range of $(c_j, c_{j+1}]$. If the invariant is preserved between each subrange, then it is preserved across the queried checkpoint range. If the AD invariant proofs for the compact subranges are succinct, then the resulting checkpoint invariant proof is also succinct. A protocol description for the $\mathsf{AHD}_{\mathsf{Amtz}}$ construction is given in Figure 5.6. We provide only the update and invariant proving logic, as the remaining functionality follows from the same history commitment and AD combination as given in Figure 5.5.

5.3.3 Security Analysis

In this section we provide theorem statements and proof sketches for the AHD security of the two generic transforms, $\mathsf{AHD}_{\mathsf{IVC}}$ (Figure 5.5) and $\mathsf{AHD}_{\mathsf{Amtz}}$ (Figure 5.6). Our AHD transforms are generic with respect to an AD, a VC, and a succinct non-interactive proof system that supports circuit relations. For the AD, we will require binding and invariant soundness, and for the VC we will require binding; defined in Figure 2.5. For the non-interactive proof system, we will require soundness.

Key binding and history binding. We first consider key binding and history binding, for which both $\mathsf{AHD}_{\mathsf{IVC}}$ and $\mathsf{AHD}_{\mathsf{Amtz}}$ use the same mechanisms. Key binding is acheived by relying directly on an underlying AD and history binding relies on an append-only VC. We provide the following theorems:

Theorem 14. *For any adversary $\mathcal{A}$ against the key binding of $\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD}, \mathsf{VC}, \mathsf{SNARK}]$, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{bind}_{\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD},\mathsf{VC},\mathsf{SNARK}],\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{bind}_{\mathsf{AD},\mathcal{B}}(\lambda)\,.$$

Theorem 15. *For any adversary $\mathcal{A}$ against the key binding of $\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD}, \mathsf{VC}]$, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{bind}_{\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}],\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{bind}_{\mathsf{AD},\mathcal{B}}(\lambda)\,.$$

Proof sketch: The lookup proofs in $\mathsf{AHD}_{\mathsf{IVC}}$ and $\mathsf{AHD}_{\mathsf{Amtz}}$ operate directly over the the AD component of the digest, d_{AD}. Thus, a win in the AHD key binding game translates directly to a win in the AD key binding game by constructing a wrapper adversary $\mathcal{B}$ that forwards the same values output from $\mathcal{A}$ replacing $d = (d_{\mathsf{AD}}, d_{\mathsf{VC}})$ with d_{AD}.

Theorem 16. *For any adversary $\mathcal{A}$ against the history binding of* $\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD},\mathsf{VC},\mathsf{SNARK}]$*, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{hbind}_{\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD},\mathsf{VC},\mathsf{SNARK}],\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{bind}_{\mathsf{VC},\mathcal{B}}(\lambda)\,.$$

Theorem 17. *For any adversary $\mathcal{A}$ against the history binding of* $\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}]$*, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{hbind}_{\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}],\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{bind}_{\mathsf{VC},\mathcal{B}}(\lambda)\,.$$

Proof sketch: We construct $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ against the index binding of VC as a relatively simple wrapper around $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$. First stage adversary $\mathcal{B}_1$ runs AHD setup and replaces the parameters for VC with its own public parameters, then runs $\mathcal{A}_1$. $\mathcal{B}_1$ parses the digest $d = (d_{\mathsf{AD}}, d_{\mathsf{VC}})$ and outputs d_{VC}.

Second stage adversary $\mathcal{B}_2$ runs $\mathcal{A}_2$ and simulates ProveHist. Whenever $\mathcal{A}_2$ queries a valid history proof, $\mathcal{B}_2$ stores the VC lookup proof for each index c_j and passes along the VC update proof to its own Prefix oracle. If $\mathcal{A}_2$ makes a query that sets the win flag, $\mathcal{B}_2$ must have two valid lookups for the same c_j index for different values, which it will return to win the index binding game. Thus, $\mathcal{B}$ wins whenever $\mathcal{A}$ wins.

Invariant soundness. The mechanisms by which invariant soundness is achieved differ between $\mathsf{AHD}_{\mathsf{IVC}}$ and $\mathsf{AHD}_{\mathsf{Amtz}}$. We consider each separately.

Theorem 18. *For any adversary $\mathcal{A}$ against the invariant soundness of* $\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD},\mathsf{VC},\mathsf{SNARK}]$*, we give adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ such that*

$$\mathbf{Adv}^{inv}_{\mathsf{AHD}_{\mathsf{IVC}}[\mathsf{AD},\mathsf{VC},\mathsf{SNARK}],\Phi_{\mathsf{vsn}},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{inv}_{\mathsf{AD},\Phi_{\mathsf{vsn}},\mathcal{B}}(\lambda) + \mathbf{Adv}^{bind}_{\mathsf{VC},\mathcal{C}}(\lambda) + \mathbf{Adv}^{\mathrm{sound}}_{\mathsf{SNARK},\mathcal{D},\mathsf{X}}(\lambda)\,,$$

where X *is the knowledge extractor for* SNARK*.*

Proof sketch: In building $\mathcal{B}$ and $\mathcal{C}$ against the invariant soundness and index binding of AD and VC, respectively, we will first need to extract the valid lookup and update proofs attested to in the recursive SNARK for the checkpoint epochs $[c_j]_j^m$. To show that valid proofs for AD and VC can be extracted, we build $\mathcal{D}$ against the soundness of SNARK that wins if this is not the case. When $\mathcal{A}$ outputs valid SNARK π_Φ, $\mathcal{D}$ extracts the full history of AD.VerUpd, VC.VerUpd, and VC.VerLkup proofs to the initial digest. It does this recursively by additionally extracting the SNARK proof for

the prior epoch, and then repeating extraction on the prior epoch SNARK proof. If any extraction fails to produce valid proofs, $\mathcal{D}$ wins the soundness game. Naively, from the way we present $\mathsf{AHD}_{\mathsf{IVC}}$, this extraction will be linear in the number of epochs. We did this for simplicity of presentation, to create a tighter reduction, one would use tree-based techniques to execute recursion with logarithmic depth [BCCT13]. Then $\mathcal{D}$ would perform logarithmic extraction for each checkpoint given by $\mathcal{A}$.

Now given these extracted proofs, we show that a winning adversary $\mathcal{A}$ corresponds to either a break in invariant soundness of AD or a break in index binding of VC. First we show that the $\mathcal{A}$ provided digests $[d_{c_j}]_j^m$ will be equal to the extracted digests (for which we have valid extracted proofs). If $\mathcal{A}$ wins, the provided digests verify under VerHist, meaning that there exists a valid VC.VerLkup proof for the digest at index c_j. On the other hand, we have a sequence of extracted VC.VerUpd proofs with an extracted lookup proof for an extracted digest each index. If the extracted digest for an index does not match that of a provided digest, we construct $\mathcal{C}$ that wins the index binding game by querying the sequence of update proofs to PREFIX and then outputting the two lookup proofs for the index c_j where the extracted digest differs from the provided digest.

Finally, now given that the provided digests match the extracted digests, we build $\mathcal{B}$ against invariant soundness. By $\mathcal{A}$'s winning condition, we have that the invariant is not preserved for key k between c_{i_A} and c_{i_B}. However, we have an extracted sequence of valid invariant proofs for AD between c_{i_A} and c_{i_B}. $\mathcal{B}$ outputs the sequence of extracted invariant proofs along with the lookup proofs provided by $\mathcal{A}$ to win the invariant soundness game for AD.

Theorem 19. *For any adversary $\mathcal{A}$ against the invariant soundness of* $\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}]$, *we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{\mathit{inv}}_{\mathsf{AHD}_{\mathsf{Amtz}}[\mathsf{AD},\mathsf{VC}],\Phi_{\mathsf{vsn}},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\mathit{inv}}_{\mathsf{AD},\Phi_{\mathsf{vsn}},\mathcal{B}}(\lambda)\,.$$

Proof sketch: In $\mathsf{AHD}_{\mathsf{Amtz}}$, the invariant proof already consists of a sequence of invariant proofs for AD between each of the checkpoints. $\mathcal{B}$ simply returns the sequence of invariant proofs between c_{i_A} and c_{i_B} along with the lookup proofs to win the invariant soundness game for AD.

5.4 Client Checkpoint Auditing

We show here how to use an AHD for the versioned invariant as described above along with a public bulletin board to build a verifiable registry. We consider a single server that maintains a dictionary

of key-value mappings within an AHD. The server collects client requests for new mappings or updates to mappings, and incorporates the updates on a regular schedule by updating the AHD and publishing, on a public bulletin board, a (signed) digest d_{i+1}, where $(d_{i+1}, st) \leftarrow \mathsf{Upd}([k_j, v_j]_j : st)$. As discussed in Section 4.2, we assume that all clients have a consistent view of this bulletin board and can efficiently lookup digests by epoch.

Client lookups, monitoring, and key updates. Clients can monitor values for keys that they own ensuring no unexpected changes have been made, or clients can lookup the value of other keys in the registry. In our construction, both actions consist of the client simply making a lookup request to the server for the desired key k. The server responds with the value v and version number u along with a proof π of the lookup for the current epoch i: $(v, u, \pi) \leftarrow \mathsf{Lkup}(k : st)$. The client reads the digest d_i from the bulletin board[4] and verifies the proof: $\mathsf{VerLkup}(d_i, k, v, u, \pi)$. If monitoring, the client additionally checks the returned value and version match the client's stored value and version. Updates to keys proceed similarly. When a client requests a key update from v to v' at epoch i, the server provides the client with a lookup proof for (v, u) in d_i and a lookup proof for the updated (v', u') incorporated in new d_{i+1}. The client, again, reads the digests from the bulletin board, verifies the proofs, and checks the version u against the stored version for the key. Finally, the client checks $u' = u + 1$ storing the new version number and value for future monitoring.

Assuming the versioned invariant is preserved between all epoch digests published to the bulletin board, these checks are sufficient for convincing a client that (1) any lookups to owned keys made by other clients returned correct values, and (2) any lookups made by the client to other keys either returned correct values or that server misbehavior will be detected the next time the key's owner performs monitoring.

Of course, the client cannot efficiently verify the versioned invariant for the full bulletin board. We solve this by requiring the client to perform a process we call *checkpoint auditing*, in which the client verifies the invariant is preserved across specific canonical checkpoint epochs. On each operation (lookup, monitoring, or update), the client performs checkpoint auditing for the epoch range (ℓ, i) where ℓ is the epoch of their last operation ($\ell = 0$ for the client's first operation) and i is

[4]We abstract away the fact that, depending on the implementation of the bulletin board, it may be convenient for the client to obtain the commitment d_i from the server, and then check consistency with the bulletin board later on. Figure 5.7 provides an optional protocol to lazily confirm consistency of server-provided digests with the bulletin board.

Protocol: Client Checkpoint Auditing

Init: The client pulls the public parameters pp_{AHD} from the registry server and verifies against the bulletin board. The client initializes its state as follows:

- (ℓ, d_ℓ): latest epoch and digest audited with registry.
- $(\ell', d'_\ell$: (optional) latest epoch and digest audited with public bulletin board.
- $T[k] = (v, u)$: table of owned keys and expected values to monitor.

Audit: Verify consistent view and invariant preservation

(1) Client computes current epoch i (deterministically computed from clock).

(2) Client computes checkpoint epochs $[c_j]_j^m$ for range (ℓ, i):

$$[(c_j, R_j)]_j^m \leftarrow \mathsf{CompactR}((\ell, i))\,.$$

(3) Client reads digests $[d_{c_j}]_j^m$ for checkpoint epochs (2 options).

(a) Client reads directly from public bulletin board.

(b) (Optional) Client reads digests from server, and lazily confirms with public bulletin board.

- Server provides checkpoint digests and history proof, which client verifies: $\pi_{\mathsf{hist}} \leftarrow \mathsf{AHD.ProveHist}(([c_j]_j^m : st_i))$.
- At some later epoch $t > i$, client reads digest d_t from the public bulletin board, and requests and verifies a history proof for checkpoints $[\ell', t]$ from the server.
- Client updates state $(\ell', d'_\ell) \leftarrow (t, d_t)$.

(4) Client requests and verifies invariant proof for checkpoints from server:

$$\pi_\Phi \leftarrow \mathsf{AHD.ProveInv}([c_j]_j^m : st_i), \qquad \mathsf{VerInv}(d_i, [(c_j, d_{c_j})]_j^m, \pi_\Phi)\,.$$

(5) Client updates state $(i, d_i) \leftarrow (\ell, d_\ell)$.

Lookup: Authenticated lookup of key k

(1) Client performs audit to current epoch i.

(2) Client requests and verifies lookup proof for audited epoch i from server:

$$(v, \pi_{\mathsf{lkup}}) \leftarrow \mathsf{Lkup}(k : st_i), \qquad \mathsf{VerLkup}(d_i, k, v, \pi_{\mathsf{lkup}})\,.$$

Monitor: Monitor owned keys in T for unexpected changes

(1) Client performs audit to current epoch i.

(2) For each $[(k_j, v_j, u_j)]_j \in T$:

(a) Client performs lookup of k_j receiving value $(\hat{v}, \hat{u})$.

(b) Client verifies $(v_j, u_j) = (\hat{v}, \hat{u})$.

Update: Update value for key k from v to v'.

(1) Server confirms update was included in epoch $i+1$.

(2) Client audits to epoch i and again from i to $i+1$.

(3) Client performs lookup of k for epoch i receiving $(\hat{v}, \hat{u})$ and verifying $(v, u) = (\hat{v}, \hat{u})$.

(4) Client performs lookup of k for epoch $i+1$ receiving $(\hat{v}', \hat{u}')$ and verifying $(v', u+1) = (\hat{v}', \hat{u}')$.

(5) Client updates $T[k] = (v', u+1)$.

Figure 5.7: Description of the continuous client auditing protocol that enables eventual inconsistency detection between clients. The registry server maintains an AHD under the versioned invariant.

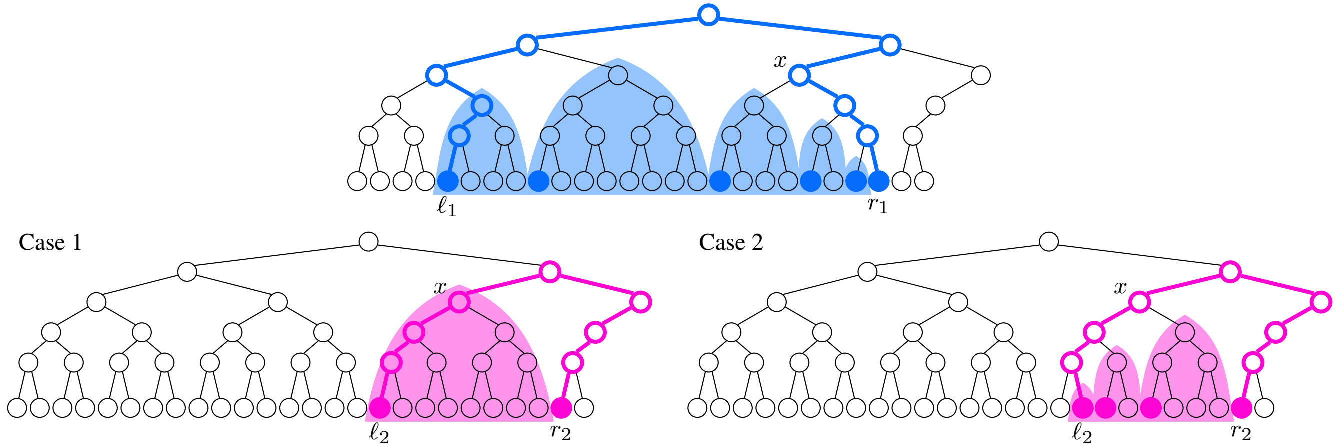

Figure 5.8: Cases for proof of shared checkpoint epoch. Case 1 (left) has ℓ_2 as leftmost leaf of x's subtree. Case 2 (right) does not.

the epoch of their current operation.

Checkpoint auditing. We make use of the notion of compact ranges from amortized proving in a different context. Clients select checkpoints $[c_j]_j^m$ for range (ℓ, i) as the endpoints in the compact range representation: $[(c_j, R_j)]_j^m \leftarrow \mathsf{CompactR}((\ell, i))$ – this results in a number of checkpoints that is *logarithmic* in the length of the range. The server proves the invariant is preserved between adjacent checkpoints, $\pi_\Phi \leftarrow \mathsf{ProveInv}([c_j]_j^m : st)$, which the client can verify after reading the checkpoint digests from the bulletin board. This is however not enough to prevent oscillation attacks (see Section 4.2). Imagine two clients auditing ranges that always result in disjoint sets of checkpoints: there will be no guarantee the invariant is preserved between digests seen by one client to digests seen by the other.

Our insight, inspired by the deterministic skiplist approach of [MB02], is summarized by the following result; a detailed pseudocode diagram of the checkpointing auditing protocol is given in Figure 5.7. The implication of this result is that two clients that individually perform checkpoint auditing will be guaranteed a shared checkpoint, and further, any deviation by the server from the invariant in the client views up until that checkpoint would have been detected.

Theorem 20. *For any two ranges (ℓ_1, r_1) and (ℓ_2, r_2) that are overlapping, i.e., $\ell_1 \leq \ell_2 < r_1 \leq r_2$, the compact range of (ℓ_1, r_1) shares a subrange boundary with the compact range of (ℓ_2, r_2). That is, for $[(\ell_{1,i}, r_{1,i})]_i^m \leftarrow \mathsf{CompactR}((\ell_1, r_1))$ and $[(\ell_{2,i}, r_{2,i})]_i^n \leftarrow \mathsf{CompactR}((\ell_2, r_2))$, there exists i, j*

such that $\ell_{1,i} = \ell_{2,j}$.

Proof. First consider the binary tree imposed over all epochs. In this binary tree, define node x as the root of the smallest subtree that contains ℓ_2 and r_1. We will show that their exists a shared boundary in the compact range representation induced by (ℓ_1, r_1) and (ℓ_2, r_2) somewhere within this subtree rooted at x.

Consider the following two exhaustive cases (depicted in Figure 5.8):

Case 1: ℓ_2 is the leftmost leaf of x's subtree.

In this case, we will show that ℓ_2 itself is a shared boundary between the two compact range representations.

If r_2 is not in x's subtree, then x's subtree (or a supertree of x's subtree if x is a left child) is included in the compact range of (ℓ_2, r_2). Otherwise, if r_2 is in x's subtree, r_2 would necessarily be in x's right subtree (since $r_2 \geq r_1$) and thus x's left subtree is included in the compact range. For the other range, since r_1 is included in x's right subtree and $\ell_1 < \ell_2$ is not in x's left subtree, then x's left subtree is included in the compact range of (ℓ_1, r_1).

Case 2: ℓ_2 is not the leftmost leaf of x's subtree.

Call y the leftmost leaf of x's right subtree. We argue that y is a shared boundary between the two compact range representations.

First, we argue that if the right endpoint of a range is in a subtree T *and* the left endpoint is not, then the leftmost leaf of T is a boundary in the range's compact range representation. Consider two cases. First, if the right endpoint is in T's right subtree, then T's left subtree is a compact subrange included in the representation and therefore, the leftmost leaf of T is a boundary. Second, if the right endpoint is in T's left subtree, then we use a recursive argument to claim that the leftmost leaf of T's left subtree (i.e., the leftmost leaf of T) is a boundary. The base case of this argument is that the right endpoint is itself the leftmost leaf of T, in which case, it is a boundary.

Now with this argument, first consider r_1 which is in x's right subtree (ℓ_1 is not), then y is a boundary of (ℓ_1, r_1). Next consider, (ℓ_2, r_2). If r_2 is in x's right subtree (ℓ_2 is not), then y is a boundary by the same argument as above. Else, r_2 is not in x's subtree, so since ℓ_2 is not the leftmost leaf of x's subtree, then x's right subtree is included as a subrange in the compact range of (ℓ_2, r_2)

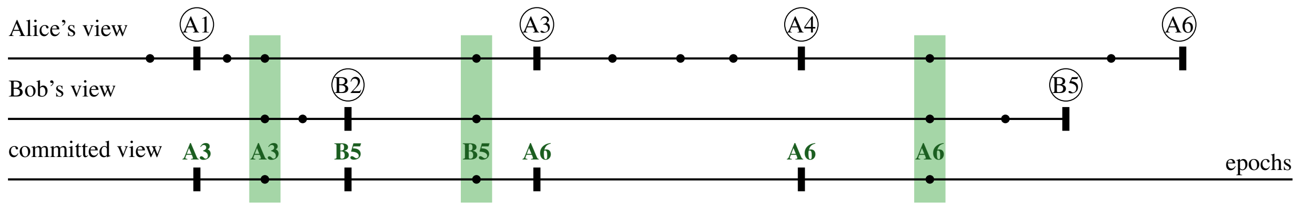

Figure 5.9: Eventual inconsistency detection for Alice's and Bob's view using shared checkpoints. Large ticks with circle labels indicate points in time where Alice or Bob perform auditing. They verify that the invariant is preserved between consecutive checkpoints selected in the range from their last audit; the checkpoints are indicated by small circles. Checkpoints are chosen to guarantee that any two of Alice and Bob's overlapping audit ranges will share at least one checkpoint, highlighted in green. Thus, the interleaved epochs at which Alice and Bob audit are implicitly guaranteed to preserve the invariant, up until their most recent shared checkpoint. The time at which an epoch is committed to their shared view is indicated on the bottom timeline. The shared checkpoint lags behind the most recent lookups made by Alice and Bob, but will eventually catch up on future lookups.

and y is a boundary. □

Consistency guarantees. The shared checkpoint progresses based on how frequently clients perform audits. More precisely, if a client is served a lookup proof that violates the invariant, it is guaranteed that one of the two clients will detect the inconsistency once each client comes online once more in sequence, i.e., if client A is served a bad lookup value, it will be detected after client B audits next and client A audits again after that. We formalize this guarantee in an *eventual detection by checkpoint auditing* security property which we prove secure for any AHD under the versioned or append-only invariant. We can illustrate the high level argument for eventual detection through a simple example illustrated in Figure 5.9.

Consider two clients, client A and client B, where client A periodically monitors a key that they own and client B performs lookups and periodic audits. Following Figure 5.9, consider the following sequence of events:

(1) Client A monitors at $A1$.

(2) Client B looks up A's key at $B2$.

(3) Client A monitors at $A3$ and $A4$.

(4) Client B audits at $B5$.

We address detection of a ghost key attack where the server serves client B a different value at $B2$ than what client A expects. Checkpoint auditing guarantees that either client A or B will detect an

inconsistency by the next time each have audited in sequence, which, in this case, is when client B audits at $B5$. We can see this by considering three ranges that were audited: (1) client B audits range $(0, B2)$ on lookup, (2) client A audits range $(A1, A3)$ on monitoring, and (3) client B audits range $(B2, B5)$. Of these three ranges, we have that $(0, B2)$ and $(A1, A3)$ are overlapping and that $(A1, A3)$ and $(B2, B5)$ are overlapping. Then by Theorem 20, we have the existence of checkpoints $C1$ and $C2$ such that invariant proofs for the following paths were checked during each audit respectively: (1) $0 \to C1 \to B2$, (2) $A1 \to C1 \to C2 \to A3$, and (3) $B2 \to C2 \to B5$. Put together, we have invariant proofs for the following path, implying that the invariant is preserved from $A1 \to B2 \to A3$:

$$A1 \to C1 \to B2 \to C2 \to A3\,.$$

Now consider for the versioned invariant, client A monitors for expected value and version (v, u) at $A1$ and $A3$. Since the invariant is preserved from $A1 \to B2$, it must be that the value (v', u') served to client B cannot be different ($v' \neq v$) unless the version number has increased $u' > u$. Similarly, from $B2 \to A3$, since the invariant is preserved, if $v' \neq v$, it must be that $u > u'$. This is a contradiction, so this inconsistency will either be caught by failure to verify client A's lookup of (v, u) during monitoring or by failure to verify an invariant proof for one of the three audits. It is clear that this argument can be extended to any pair of clients.

Lastly, we note the interplay between checkpoint auditing and $\mathsf{AHD}_{\mathsf{Amtz}}$. Since the format of checkpoints that are passed to $\mathsf{ProveInv}$ are already compact subranges, the invariant proof for each pair of checkpoints consists of a single precomputed proof, instead of a logarithmic sequence of proofs. This results in proof sizes for checkpoint auditing to be of size $\mathcal{O}(\log N)$ as opposed to $\mathcal{O}(\log^2 N)$ for range length N.

More formally, we provide a formal definition for checkpoint auditing with respect to an authenticated history dictionary and an immediately consistent bulletin board. This definition is inspired by the "oscillation" security definition of [MKL+20] in which an adversary wins if a client does not detect a ghost key attack. Our security game is slightly more complex as client checkpoint auditing requires pairwise clients to perform audits, in contrast to [MKL+20] where a single client may audit and rely on additional assurances from trusted third-party auditors.

The security game for detection of invariant breaks in checkpoint auditing is defined by the pseudocode game CKPTDETECT given in Figure 5.10. It models two clients: client 1 monitors an

Game $\textsc{CkptDetect}^{\mathcal{A}}_{\mathsf{AHD},\Phi}(\lambda)$

$pp \leftarrow\!\!\$\ \mathsf{AHD.Setup}(\lambda)$
$(d_0, st) \leftarrow \mathsf{AHD.Init}()$
$\ell \leftarrow 0\,;\ \ell_0 \leftarrow 0\,;\ \ell_1 \leftarrow 0$
$v \leftarrow \bot\,;\ V \leftarrow [\cdot]$
$aud_0 \leftarrow [\cdot]\,;\ aud_1 \leftarrow [\cdot]$
$k \leftarrow\!\!\$\ \mathcal{A}_1(pp)$
$(i_1, i_2, i_3, v', \pi_{\mathsf{lkup}}) \leftarrow\!\!\$\ \mathcal{A}_2^{\textsc{PubDigest},\textsc{Audit}_0,\textsc{Monitor}_1,\textsc{UpdVal}_1}$

$$\text{Return } \bigwedge \begin{pmatrix} i_1 \le i_2 \le i_3 \\ i_1 \in aud_0 \\ i_2 \in aud_1 \\ i_3 \in aud_0 \\ \mathsf{AHD.VerLkup}(d_{i_1}, k, v', \pi_{\mathsf{lkup}}) \\ v' \ne V[i_1] \end{pmatrix}$$

Oracle $\textsc{PubDigest}(d)$

$V[\ell] \leftarrow v$
$\ell \leftarrow \ell + 1$
$d_\ell \leftarrow d$

Oracle $\textsc{Audit}_0(\pi_\Phi, \pi_{\mathsf{hist}})$

Require $\ell > \ell_0$
$\left[(c_j, R_j)\right]_j^m \leftarrow \mathsf{CompactR}(\ell_0, \ell)$
Require $\mathsf{AHD.VerInv}(d_\ell, [(c_j, d_{c_j})]_j^m, \pi_\Phi)$
Require
$\mathsf{AHD.VerHist}(d_\ell, [(c_j, d_{c_j})]_j^m, \pi_{\mathsf{hist}})$
$\ell_0 \leftarrow \ell\,;\ aud_0 \leftarrow aud_0 \,\|\, [\ell]$

Oracle $\textsc{Monitor}_1(\pi_\Phi, \pi_{\mathsf{hist}}, \pi_{\mathsf{lkup}})$

Require $\ell > \ell_1$
$\left[(c_j, R_j)\right]_j^m \leftarrow \mathsf{CompactR}(\ell_1, \ell)$
Require $\mathsf{AHD.VerInv}(d_\ell, [(c_j, d_{c_j})]_j^m, \pi_\Phi)$
Require
$\mathsf{AHD.VerHist}(d_\ell, [(c_j, d_{c_j})]_j^m, \pi_{\mathsf{hist}})$
Require $\mathsf{AHD.VerLkup}(d_\ell, k, v, \pi_{\mathsf{lkup}})$
$\ell_1 \leftarrow \ell\,;\ aud_1 \leftarrow aud_1 \,\|\, [\ell]$

Oracle $\textsc{UpdVal}_1(\pi_\Phi, \pi_{\mathsf{hist}}, \pi_{\mathsf{lkup}}, v')$

Require $\ell = \ell_1 + 1$
Require $\mathsf{AHD.VerInv}(d_\ell, [(\ell_1, d_{\ell_1})], \pi_\Phi)$
Require $\mathsf{AHD.VerHist}(d_\ell, [(\ell_1, d_{\ell_1})], \pi_{\mathsf{hist}})$
Require $\mathsf{AHD.VerLkup}(d_\ell, k, v', \pi_{\mathsf{lkup}})$
$v \leftarrow v'$
$\ell_1 \leftarrow \ell\,;\ aud_1 \leftarrow aud_1 \,\|\, [\ell]$

Figure 5.10: Security game for invariant break detection with checkpoint auditing.

adversary-chosen key over time, and client 0 performs lookups to the key. The goal of the adversary is to serve a lookup value that is accepted by client 0, but is not consistent with the "true" value maintained by client 1. More specifically, the adversary may induce periodic audits by client 0 and periodic monitoring audits or value updates of the key by client 1. The exposed oracles represent the verification procedure that clients would take during auditing and, as such, take as input proofs that would be served by the server (fully controlled by the adversary). As specified in Figure 5.7, clients audit logarithmic number checkpoints selected by the compact range, and on a value update, a monitoring client audits up to epoch prior to the change in value. The adversary also has full control over the digest, but may only publish a single digest for each epoch, representing an immediately consistent bulletin board. At the end of the game, the adversary outputs a value v' and lookup proof along with three epoch numbers i_1, i_2, i_3. The adversary wins if the following conditions are satisfied:

(1) v' verifies under the lookup proof at epoch i_1 for client 0. Thus, we additionally require that client 0 audited epoch i_1.

(2) v' does not match the expected value of the key at digest i_1 as monitored by client 1 and tracked by the game.

(3) The appropriate eventual detection auditing conditions have been met. There exists an epoch $i_2 \geq i_1$ that was monitored by client 1, and there exists an epoch $i_3 \geq i_2$ that was audited by client 0.

We define an adversary $\mathcal{A}$'s advantage against the checkpoint auditing game as:

$$\mathbf{Adv}^{\mathsf{ckpt}}_{\mathsf{AHD},\Phi,\mathcal{A}}(\lambda) = \Pr[\textsc{CkptDetect}^{\mathsf{AHD},\Phi}_{\mathcal{A}}(\lambda) = 1]\,.$$

We provide the following theorem for the checkpointing security of any AHD:

Theorem 21. *For any adversary $\mathcal{A}$ against the checkpoint auditing eventual detection of* AHD*, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}^{\mathit{ckpt}}_{\mathsf{AHD},\Phi_{\mathsf{vsn}},\mathcal{A}}(\lambda) \leq \mathbf{Adv}^{\mathit{inv}}_{\mathsf{AHD},\Phi_{\mathsf{vsn}},\mathcal{B}}(\lambda)\,.$$

Proof sketch: Say the expected value of k at i_1 according to client 1 is v (i.e., $V[i_1] = v$), and further say that it was updated to be equal to v' at epoch $i_0 \leq i_1$. Redefine i_2 to be the smallest epoch in aud_1 such that $i_2 \geq i_1$. This ensures that $V[i_2] = v$, and it is guaranteed that such an i_2 redefinition exists by how V is populated in $\textsc{CkptDetect}$. For a winning adversary, we have that the following sequences of invariant proofs were verified by either client 0 or client 1 because i_1, i_2, i_3 are in the sets of successful audits:

(1) $0 \to c_1 \to i_1$: We know that client 0 verified some path of valid invariant proofs from epoch 0 through a shared checkpoint c_1 (we will show why such a shared checkpoint exists shortly) through epoch i_1.

(2) $i_0 \to c_1 \to c_2 \to i_2$: We know that client 1 verified some path of valid invariant proofs from i_0 to i_2, and since (i_0, i_2) is overlapping with $(0, i_1)$, by Theorem 20, we have the existence of shared checkpoint c_1, where $i_0 \leq c_1 \leq i_1$. Again, we will show shortly the existence of a second shared checkpoint c_2.

(3) $i_1 \to c_2 \to i_3$: We know that client 0 verified some path of valid invariant proofs from i_1 to i_3, and since (i_1, i_3) is overlapping with (i_0, i_2), by Theorem 20, we have the existence of shared checkpoint c_2, where $i_1 \leq c_2 \leq i_2$.

Taking the above, we have that between client 0 and client 1, a path of valid invariant proofs were

verified for:

$$i_0 \to c_1 \to i_1 \to c_2 \to i_2 \, .$$

By our redefinition of i_2, we have that client 1 verified lookup proofs for v at i_0 and i_2. By the adversary's winning condition, we have that client 0 verified a lookup proof for v' at i_1. However, by the versioned invariant, it is not possible for the value to change to v' and then change back to v. Say $\mathsf{v} = (v, u)$ where u denotes the version number. By the versioned invariant, $\mathsf{v}' = (v', u')$, where $u' > u$. Any future value $\mathsf{v}'' = (v'', u'')$ that preserves the versioned invariant cannot have $u'' < u'$, so a valid lookup of $\mathsf{v} = (v, u)$ at epoch i_2 is a break in invariant. We build $\mathcal{B}$ to output the set of invariant and history proofs from epoch i_1 to epoch i_2 and provide the lookup proof at i_1 for v' output by $\mathcal{A}$ at game end and the lookup proof at i_2 provided by $\mathcal{A}$ to $\textsc{Monitor}_1$ at i_2 for v.

5.5 Evaluation

5.5.1 Implementation

We implement our proposed constructions in Rust. Our implementation consists of a number of modular parts (following Figure 5.1). We define a generic authenticated dictionary interface that supports versioned invariant update proofs for consecutive epochs, and an accompanying interface for generating SNARK constraints for verification of the update proof. We then implement our two generic transforms, IVC (Figure 5.5) and amortized proving (Figure 5.6), to take an object implementing the authenticated dictionary interface and produce an object implementing a defined authenticated history dictionary interface. Lastly, given an object implementing the AHD interface, we instantiate a verifiable registry service exposing a RESTful API for key lookups, key updates, and client checkpoint auditing (Figure 5.7). The service is backed by an in-memory Redis datastore. In total, our implementation consists of $\approx$ 12000 lines of code and is available open source at `https://github.com/nirvantyagi/versa`.

The constraints and generic IVC transform are implemented within the `arkworks` ecosystem for SNARKs [BCG+20b] and make use of the SNARK implementations from `arkworks`. We instantiate and evaluate the recursion constructions on [Gro16] over the MNT4-753 and MNT6-753 pairing-friendly cycle of curves to target 128 bits of security. This choice of SNARK requires a trusted

setup and results in a state-of-the-art constant proof size; however, other general-purpose recursive SNARKs [Set20, CHM+20, BCMS20, BDFG21] can be swapped in with different trade-offs in setup assumptions, proving costs, and proof size. Ultimately, looking forward to evaluation, we will be interested in the difference between SNARK proving costs for verifying the Merkle tree AD update proof versus the RSA AD update proof. We expect the proving cost ratio to be comparable across SNARKS as it is dependent on the ratio of circuit constraints.

VeRSA constructions. We build our two VeRSA variants using the described modular implementation. First we implement the KVaC RSA AD [AR20] along with our proposed update proof (Section 5.2) following the proof of homomorphism over hidden order groups [BBF19]. We instantiate the construction with an RSA group of 2048 bits. We further implement SNARK constraints for verification of the update proof; the constraints make use of optimizations for multiprecision arithmetic [KPS18] and hashing to primes [OWWB20]. VeRSA-IVC is the registry resulting from the modular IVC transform and VeRSA-Amtz is the registry from the amortized proving transform. Our RSA constructions require a hidden-order RSA group from a trusted setup; while not ideal, academic work [CHI+21, BGG18, BGM17] has suggested that large-scale multi-party setup ceremonies can be conducted in practice. Class groups [BW88] provide an alternate tack to constructing a hidden-order group without trusted setup, but would significantly hinder performance.

Baselines. To evaluate our VeRSA constructions, we compare to verifiable registries based on Merkle tree ADs. We implement a Merkle tree AD supporting versioned invariant proofs. The first baseline, which we denote as MT-VR, is the verifiable registry not designed for efficient client auditability in which update proofs for each consecutive epoch must be checked, either by the client or a trusted auditor party. The performance characteristics of MT-VR represent a set of previous work, most closely being CONIKS [MBB+15], but also sharing structure with SEEMless [CDGM19] and Mog [MKL+20]. The second baseline we consider is the verifiable registry resulting from applying the IVC transform to the Merkle tree AD, which we denote MT-VR-IVC. While we use this construction as a baseline, as it has been proposed abstractly in concurrent work [CCDW20], we note, to the best of our knowledge, ours is the first implementation of this approach. We set the height of the Merkle tree to 32, which with our open addressing optimization can support 2^{30} keys, and instantiate the hash function with the Poseidon algebraic hash function [GKK+19] for MT-VR-IVC

and with SHA3 for MT-VR.

Experimental setup. We wish to answer the following questions about VeRSA-IVC and VeRSA-Amtz in comparison to the Merkle tree baselines:

- *Client auditing costs:* What are the bandwidth and computation costs for a client to audit a range of epochs?
- *Server update costs:* What are the computation costs for the server to incorporate key updates and publish a new epoch digest? At what latency can new digests be published; supporting what key update throughput?
- *Lookup costs:* What are the bandwidth and computation costs for key lookups?

We benchmark our constructions using an Amazon EC2 `r5.16xlarge` instance with 32 CPU cores and 512 GB memory. Client computation is evaluated single-threaded, and network costs of gathering client input are not evaluated; our experiments simulate client input, generating random requests of the appropriate size.

VeRSA-Amtz grows in update cost over the history of the registry due to increasing amortized costs of proving. We present the amortized costs of proving for epoch 2^k by averaging the proving costs incurred between the 2^{k-1} updates from epoch $2^{k-1}+1$ to 2^k. While these proving costs occur in spikes over the range, VeRSA-Amtz is not delayed by the need to complete an expensive proof for a large range; the invariant proofs can be computed in the background and audits can still be fulfilled (see further discussion on parallelism in Section 5.5.3). Therefore, we believe reporting the amortized costs in this manner leads to a fair evaluation.

5.5.2 Client Auditing Costs

We contrast the auditing costs in terms of proof size and verification time for different lengths of audit ranges; the results are shown in Figure 5.11. MT-VR-IVC, VeRSA-IVC, and VeRSA-Amtz have client auditing costs that scale logarithmically in the length of the audit range. Note, the IVC constructions' proof size and verification costs would become truly constant were they instantiated in the auditing model where a third-party verifies the hashchain. In any case, the costs among the client-auditable constructions, VeRSA and MT-VR-IVC, are comparable. The proof sizes, even for

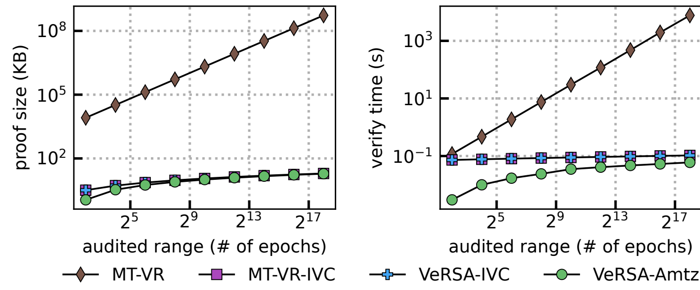

Figure 5.11: Client auditing costs. The size (left) and verification time (right) of invariant proofs for varying epoch range lengths.

large epoch ranges, remain under 20 KB, and proofs are verified in under 100 ms.

The naive comparison for client auditing costs is the baseline MT-VR in which clients (or trusted auditors) must perform linear work auditing every consecutive epoch. Against MT-VR, for an epoch range of length 32, client bandwidth costs are $10^3\times$ lower and verification time is $10\times$ lower. For epoch ranges of length 1000, the improvement grows to $10^5\times$ lower for bandwidth costs and $10^3\times$ lower for verification time. In context, with an epoch publishing time of 5 minutes, auditing at epoch ranges of length 32 and 1000 correspond to a client auditing every 3 hours or twice a week, respectively.

5.5.3 Server Epoch Update Costs

Building efficiently auditable proofs for clients adds significant computational costs to the server. We investigate what levels of key update throughput are achievable and at what latency. To anchor our evaluation, we set a target of ≈ 60 key updates per second based on current statistics from the certificate transparency ecosystem.

Figure 5.12 shows the latency to prove an epoch update depending on how many key updates are made in the epoch. The throughput is computed as the number of key updates divided by latency. At a high level, we find that VeRSA-IVC and VeRSA-Amtz can both achieve throughput levels > 60 key updates per second, while MT-VR-IVC achieves only ≈ 1 key update per second under the tested computation resources; we discuss how throughput can be increased through increased parallelism later.

However, for VeRSA-IVC to achieve a throughput of 60 key updates per second, epochs are

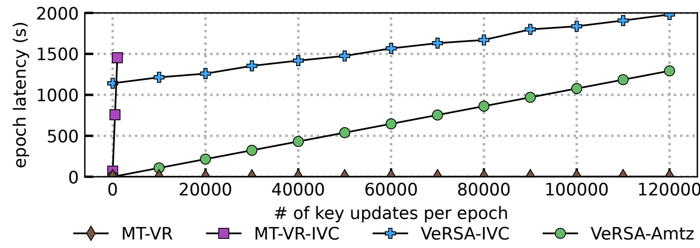

Figure 5.12: Server epoch update costs plotting the epoch update latency varying the number of key updates batched in the epoch. The key update throughput is computed as the number of key updates per epoch divided by the epoch latency. MT-VR-IVC measurements are truncated due to running out-of-memory on the benchmark machine.

published at a latency of ≈ 30 minutes. This is because of the large constant cost of verifying the RSA AD update proof within a circuit. This cost is incurred per epoch update no matter how many key updates are included, but the incremental cost of including more key updates is minimal as they do not increase the dominating cost of proving the SNARK. Thus, the throughput of VeRSA-IVC increases when more key updates are batched together. At its limit, we can extrapolate from our experiments that the throughput will cap out at ≈ 400 key updates per second due to costs of performing RSA exponentiation and computing the algebraic invariant proof (outside of the SNARK).

On the other hand, the throughput of VeRSA-Amtz is not affected by the number of key updates in an epoch; the latency is directly proportional to the number of key updates. VeRSA-Amtz achieves a throughput of ≈ 90 key updates per second while supporting publishing digests at low latencies. So while VeRSA-IVC can achieve higher throughput than VeRSA-Amtz, it would require a significantly higher latency that may not be suitable for some deployments — extrapolated results indicate VeRSA-IVC to surpass VeRSA-Amtz in throughput at a latency of 50 minutes. In contrast to MT-VR, which achieves a throughput of $40,000$ key updates per second but does not produce efficiently auditable proofs, our VeRSA systems incur a $\approx 480\times$ proving overhead.

Lastly, in terms of persistent storage, VeRSA-Amtz incurs 1123 B per epoch, and VeRSA-IVC and MT-VR-IVC incur (on average) just 64 B per epoch for the history tree vector commitment.

Improving throughput via parallelism. The dominant cost for the IVC constructions (VeRSA-IVC and MT-VR-IVC) is the SNARK proving time, and it has been shown that SNARK proving work is highly parallelizable [WZC$^+$18]. Thus, we would expect the throughput of the IVC constructions to increase more-or-less directly with increased computation resources. Figure 5.13 (left) shows

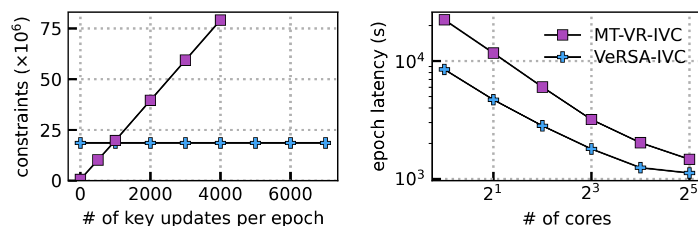

Figure 5.13: (Left) The number of constraints in the SNARK circuit for varying number of key updates. (Right) The epoch latency (dominated by the SNARK proving time) for different levels of hardware parallelism.

the number of constraints to be proved in the SNARK circuit for different numbers of key updates batched per epoch. The RSA circuit is of constant size, just under 20M constraints. The MT circuit grows linearly with the number of key updates, $\approx 20,000$ constraints per key update. We demonstrate the parallelism of the workload by measuring epoch update latency using different numbers of physical cores, shown in Figure 5.13 (right). For the circuit sizes evaluated, doubling the number of processors halves the epoch latency up until between 16 and 32 processors where the marginal benefits of adding more processors decreases. Larger circuit sizes, e.g. by adding more key updates to the MT constructions, will continue to benefit from increased processors [WZC$^+$18].

In VeRSA-Amtz, the dominant cost consists of proving invariant proofs for large subranges over the registry's life. While proving a single invariant proof (Wesolowski proof of homomorphism [Wes19, BBF19]) is mostly a sequential task, at any one time there will be approximately $\log N$ (for N total epochs) such invariant proofs being proved in the background, one for each subrange length. These tasks can be easily parallelized given $\log N$ processors such that the epoch update cost for VeRSA-Amtz is constant instead of logarithmically increasing over time. It is reasonable to assume computational resources supporting $\log N$ parallelism. For example, in our experiments with 32 cores, it would take a registry publishing epochs at 5 minute latency thousands of years to reach 2^{32} epochs.

Improving throughput via sharding. A second way to increase throughput is by sharding the key space and running separate instances of a verifiable registry. If perfectly sharded, i.e., key updates are evenly distributed across shards, then the throughput of the system is expected to increase proportionally to the number of shards (assuming the total computing resources are also increased proportionally). However, client auditing costs will increase proportionally: clients must audit each

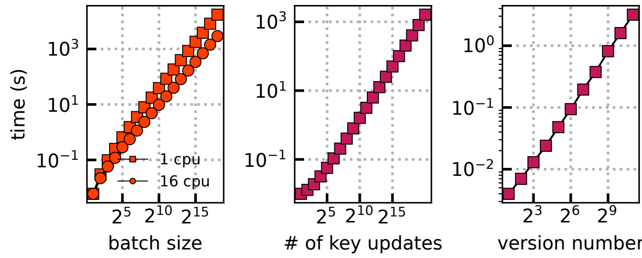

Figure 5.14: (Left) Batch computation of RSA membership proofs for varying levels of hardware parallelism. (Middle) Update computation of an individual RSA membership proof over a range of key updates. (Right) Verification costs of RSA membership proofs with respect to version number of entry.

shard assuming keys are distributed randomly across shards. If we can guarantee that each client will only be interacting with a small number of shards, then the throughput gains of sharding may come with little increase in client cost.

5.5.4 Key Lookup Costs

The VeRSA constructions achieve higher key update throughput than Merkle tree solutions, however they also incur large costs for computing membership proofs for key lookups. In the full version [TFZ+21], we describe techniques for batch membership proof computation to manage these costs. We evaluate these costs and find that VeRSA can reasonably compute lookup proofs for registries storing on the order of millions of keys, however for hundreds of millions or billions of keys, the costs of producing timely lookup proofs are infeasible. In contrast, producing lookup proofs for Merkle tree registries is extremely low cost (order of milliseconds) even for registries with billions of keys.

Figure 5.14 (left) shows the time to compute all membership proofs for a batch of keys. As a concrete example, consider a registry with 1 million keys: Figure 5.14 indicates that membership proofs for all keys can be computed using a single thread every ≈ 3 hours. In the time between batch computations, membership proofs become outdated as the registry updates, and if queried must be updated individually. Figure 5.14 (right) shows the cost of updating a single membership proof with respect to the number of key updates made to the registry. With an update throughput of 60 key updates per second, in the 3 hour batch update cycle of our example, $\approx 2^{13}$ key updates are made, which can be individually applied to respond to a lookup proof in ≈ 10 seconds. This strategy

does not incur any storage overhead on top of the storage of the lookup proofs themselves. More advanced caching strategies for batch updating frequently queried keys may be employed to further improve lookup costs. Nonmembership proofs for lookups of all possible non-member keys cannot be precomputed efficiently and must be responded to in a delayed fashion by batch computation of a set of non-member lookups together on some schedule.

Lookup proofs are small and of constant size: 0.8 KB for VeRSA, comparable to the 1 KB proof sizes of MT-VR. Figure 5.14 plots the verification time of a lookup proof with respect to the version number. Verification increases linearly with version number because the verifier must compute an RSA exponentiation to an exponent of the form $\mathsf{H}(k)^u$. Despite this trend, we find that the cost of verification remains feasible for clients if version numbers do not get too large (< 1 second for version numbers less than 1000). We believe this range of version numbers is reasonable for our envisioned applications of binary transparency and PKI for E2EE messaging. As an example for a potential application of binary transparency, we crawled version history for a random sample of 1000 software packages available in the Ubuntu 22.04 main repository. These packages had a mean of 3.4 versions (median 3), with a maximum value of 20. We also manually recorded the version history of the ten most popular apps on the iOS App Store finding a median of 52.5 (maximum of 127) versions published in 2021. If a setting must support large version numbers u, we provide details for a dictionary variant that increases lookup proof size by $\log_t u$ for some branching factor t but allows for verification in time $\log_t \times$ the time to verify a constant-size lookup proof of version t (see the full version [TFZ+21]).

CHAPTER 6

ETHICS DISCUSSION

The promise of end-to-end encryption is to ensure that the messaging platform cannot read nor interfere with conversations between its users. Unfortunately these same properties hamper the ability of the platform to perform critical safety services on behalf of its users. We have seen in WhatsApp the negative effects of safety systems being unable to see message content directly: hate speech and political disinformation campaigns are a few examples of abuse evading moderation found on WhatsApp. Such failings have led to calls for the rollback of end-to-end encrypted messaging and highlight an ongoing tension between communicating in private and holding communications accountable online.

In this chapter, we examine this tension through the lens of competing values: most pertinently, the values of privacy and accountability. Nissenbaum famously argued for the notion of "privacy in public" [Nis17]. This work will concern itself with a related notion one might call "accountability in private". It will be important to distinguish, in this chapter, the use of the term *privacy* when it refers to the value versus when it refers to descriptive properties of communication such as confidentiality, deniability, and anonymity. We will use *privacy* to refer to the former and describe communications *in private* to refer to the latter. Further, we will focus on one-to-one messaging as a starting application which provides a good case study for understanding the privacy-accountability tension while avoiding additional complexities of group messaging and more expressive social media.

First to illustrate the competing values, let us draw an analogy to the common example pertaining to privacy of a family within their home. Just as a family has legitimate claims to privacy within their home, users of messaging platforms have claims to privacy for their communication. At the same time, misbehavior of bad actors, for example, domestic abuse within the home or propagation of hate speech online is behavior that does not have a legitimate claim to privacy, and rather, the bad actor should be held accountable for such actions. These two values come into conflict when mechanisms needed to hold bad actors accountable would at the same time be violating the privacy of good actors by, for example, subjecting them to undue surveillance.

Motivating this work are two ongoing trends in the usage of online communication systems. Both of these trends exert destabilizing forces on society's existing norms around one-to-one messaging.

There are two such norms which we describe by their deployment setting. The first is plaintext messaging in which messages sent between users can be read and processed by the platform. The second setting, in the early stages of its adoption, is end-to-end encrypted messaging in which messages sent between users are encrypted using keys held only by the communicating partners, preventing the platform from learning the communication content. In the plaintext setting, platforms are trusted to view user messages and intervene to protect users from certain types of spam and abuse. In the encrypted setting, platforms generally perform few safety services on behalf of users.

Now consider the first destabilizing trend. Internet companies owning messaging platforms are increasingly collecting sensitive data about users and using that data for purposes not aligned with user interests, e.g., targeted advertising; this represents a threat to the value of privacy on plaintext systems. In contrast, the second trend considers accountability. Online messaging has always harbored bad actors, but the growth in popularity of online messaging and its increased usage as a primary form of communication has made it an expedient target for nefarious activity. Recent years have exposed coordinated attacks hosted on messaging platforms leading to real and amplified societal harm. Mitigations for these types of abuses are actively being developed for the plaintext setting but do not translate to the encrypted setting due to a lack of information flow needed to enable the abuse mitigation systems; this represents a threat to the value of accountability and safety on encrypted systems. These trends point towards an ongoing shift in norms, as evidenced by the significant amount of public discourse on the topic, and motivates new techniques to address the conflicting values.

As a pluralist society that subscribes to different values, it is inevitable that there will be conflicts and we will have to decide on appropriate recourse [Ber13]. Flanagan and Nissenbaum [FN14] offer three paths: (1) Society may trade-off one value for the other choosing to satisfy one but not the other; (2) Society may balance values compromising and appealing to each value in some limited capacity; or (3) Society may resolve the tension by changing the parameters of the setting. In the example of family privacy in the home above, society employs a complex balancing solution encoded in our legal system with certain parties such as judges and police entrusted with special privileges.

In this dissertation, we aim to dissolve the tension for certain settings in online communication by introducing new technological tools in the form of cryptographic protocols. Modern cryptography can be used to enforce precise and restricted information flows under specified conditions. We

propose new protocols to address a number of different types of abuse and associated mitigations seen on online messaging. However, as we will see, even with these new protocols, limitations in the setting or cryptography will mean that the full tension is not dissolved. In these cases, we will again appeal to balancing and seek out an appropriate compromise for the conflicting values. The following sections will discuss each dissertation chapter in turn, discussing the values at play and the manner in which the proposed technology addresses conflicting values.

Secure reporting for content moderation. In this chapter, we proposed the use of cryptographic reports for enabling content moderation. Specifically, the key insight for addressing the privacy and accountability conflict is that information needed to enable content moderation does not need to be made available to the moderator until a user report identifying the misbehavior is made. Until that time, communication content is cryptographically hidden, and users have the ability to provide cryptographic key material in a report to unlock the necessary information for the moderator to effectively moderate.

More formally, we may codify the above as an information flow in the language of contextual integrity [Nis04]. There does not exist a "one-size-fits-all" description for the information flow as it will vary depending on the type of abuse being addressed; in this work, we discuss two examples, reporting for sender abuse, and reporting for viral forwarding abuse. We will describe the information flow to generically capture both settings. The sender of information is the reporting user. The recipient is the moderator of the platform. The moderator is a special party entrusted to maintain the safety of the platform. The subject of the flow is variable depending on the type of abuse being addressed and the information needed to mitigate the abuse. For reporting sender abuse, the subject is the sender of the message. For reporting viral forwarding abuse, the subject may consist further of the source sender of the message or even the identities of all users affected by the forwarded message. Similarly, the transmission principle will also depend on the abuse type. Ideally, the flow should only be transmitted to the moderator if an abuse has occurred. In the user reporting approach, we approximate this by transmitting when a user detects and reports abuse to the moderator.

Prior to our work, such a transmission principle could not be enforced. In the plaintext setting, the moderator would have access to all the information regardless of whether an abuse had occurred. Here, the moderator as a special party was simply trusted to not abuse this power. In contrast, in

the encrypted setting, there was no mechanism for users to report content to the moderator. In fact, messaging systems were designed to explicitly prevent this type of reporting by supporting *deniability* meaning any report a user may make claiming that an abusive message was sent by a sender could have just as easily been forged by the reporting user themself.

In our work, cryptographic protocols are developed to carefully enforce the specified transmission principle and subject. That is, the relevant information is only transmitted to the moderator upon a report from a user; and even then, the information remains deniable to any party other than the moderator. This protocol removes additional trust assumptions needed for the special moderator party.

Even so, our user reporting approach makes some compromises. The transmission principle relies on a user report, but what if a user abuses their reporting power and reports non-abusive messages that should not be revealed to the platform moderator? This concern is especially relevant to the case of viral forwarding where a report may erroneously reveal a large amount of information to the platform (some of which may not have been privy even to the reporting user). Here again, moderators will need to be trusted to not abuse such erroneous reports. In future work, cryptography may again be used to help further minimize these compromises. Transparency mechanisms may be enacted to disincentivize moderators from acting on erroneous reports or more advanced transmission principles may be enforced. For example, instead of transmitting on a single user's report, transmission only occurs following a threshold number of user reports.

Sender-anonymous blocklisting. In this chapter, we develop a protocol for sender-anonymous blocklisting where the blocklist is controlled and held in private to the recipient user. One might argue that the platform can determine which senders are "safe" and protect users from spam and other abuse through other mechanisms, filtering or otherwise. However, we appeal to the value of autonomy to justify providing blocklisting powers to users; they may choose to block others based on concerns for safety or they may choose to block based on other reasons. Importantly, if user-chosen blocklists are made public to the platform, such information is sensitive and may influence users into making different blocklisting decisions, hindering their autonomy. On the other hand, prior to our work, it was not known how to do platform-level blocking with a hidden blocklist in anonymous messaging, raising a conflict between privacy and autonomy. Our work introduces new cryptography

that dissolves this conflict showing it is possible to support both together.

Verifiable public key infrastructure. This chapter investigates the trust assumptions needed to enable communications in private for encrypted messaging. When discussing trust assumptions, it is important to distinguish between the trust assumptions in an abstracted cryptographic protocol (i.e., mathematical adversaries) and the sociotechnological trust assumptions that arise from today's modern computing ecosystem (i.e., open source review, binary build pipelines, etc.) [BNP23]. This work considers the cryptographic trust assumptions.

When formalizing privacy in terms of contextual integrity and information flow, there often exists the notion of special parties that are entrusted with certain privileges. As Nissenbaum discusses, a conflict of interest between parties does not lend itself to a stable trust relationship [Nis01]. For distribution of public keys, users are required to trust the platform to enable communications in private. This trust relationship is tenuous. As discussed earlier, a motivating trend of this work is the overstepping of platforms in collecting sensitive information about users. In this chapter, new cryptography is developed to rearrange these trust dependencies so that users may audit the key distribution process hosted by the platform.

CHAPTER 7

CONCLUDING THOUGHTS

Looking forward, I want to ensure that most user interactions with internet services, even beyond messaging, are both private and accountable by default. Much progress has been made over the past decade, but there is a lot to do to realize this vision. As the internet gets increasingly converted to privacy-preserving technologies, accountability issues will continue to arise in emerging applications. Communication and messaging are a low level functionality that will be built on to create highly expressive privacy-preserving applications. Supporting these applications will require fundamentally rethinking many parts of our internet architecture and developing new cryptography.

BIBLIOGRAPHY

[ABM15] Michel Abdalla, Fabrice Benhamouda, and Philip MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE S&P*, 2015.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle diffie-hellman assumptions and an analysis of DHIES. In *CT-RSA*, 2001.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT*, 2019.

[AFG+10] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-preserving signatures and commitments to group elements. In *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 209–236. Springer, 2010.

[AKTZ17] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security*, 2017.

[AM18] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A practical system for binary transparency. 11025:94–110, 2018.

[AR20] Shashank Agrawal and Srinivasan Raghuraman. KVaC: Key-value commitments for blockchains and beyond. In *ASIACRYPT (3)*, volume 12493 of *Lecture Notes in Computer Science*, pages 839–869. Springer, 2020.

[AS16] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[ASB+17] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *IEEE Symposium on Security and Privacy*, pages 137–153. IEEE Computer Society, 2017.

[BBF19] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 561–586. Springer, 2019.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, 2004.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120. ACM, 2013.

[BCD+17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *EuroS&P*, pages 301–315. IEEE, 2017.

[BCG+20a] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *IEEE S&P*, 2020.

[BCG+20b] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2020.

[BCK+14] David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *CCS*, pages 382–393. ACM, 2014.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2020.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294. Springer, 2014.

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In *CRYPTO (1)*, volume 12825 of *Lecture Notes in Computer Science*, pages 649–680. Springer, 2021.

[Ben87] Josh Daniel Cohen Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, USA, 1987.

[Ber13] Isaiah Berlin. *The crooked timber of humanity: Chapters in the history of ideas*. Princeton University Press, 2013.

[BF99] Dan Boneh and Matthew K. Franklin. An efficient public key traitor tracing scheme. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 1999.

[BFR+13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *SOSP*, pages 341–357. ACM, 2013.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, 2004.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. In *Financial Cryptography Workshops*, volume 10958 of *Lecture Notes in Computer Science*, pages 64–77. Springer, 2018.

[BGJP23] James Bartusek, Sanjam Garg, Abhishek Jain, and Guru-Vamsi Policharla. End-to-end secure messaging with traceability only for illegal content. In *EUROCRYPT (5)*, volume 14008 of *Lecture Notes in Computer Science*, pages 35–66. Springer, 2023.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptol. ePrint Arch.*, 2017:1050, 2017.

[BKLZ20] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. In *IEEE Symposium on Security and Privacy*, pages 928–946. IEEE, 2020.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT*, 2003.

[BNP23] Ero Balsa, Helen Nissenbaum, and Sunoo Park. Cryptography, trust and privacy: It's complicated. In *CSLAW*. ACM, 2023.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, 2003.

[Bon16] Joseph Bonneau. Ethiks: Using ethereum to audit a CONIKS key transparency log. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 95–105. Springer, 2016.

[BP04] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2004.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[BS04] Dan Boneh and Hovav Shacham. Group signatures with verifier-local revocation. In *CCS*, 2004.

[BS17] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2017. Version 0.4.

[BSZ05] Mihir Bellare, Haixia Shi, and Chong Zhang. Foundations of group signatures: The case of dynamic groups. In *CT-RSA*, 2005.

[But] Vitalik Buterin. The dawn of hybrid layer 2 protocols. `https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html`.

[BW88] Johannes Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *J. Cryptol.*, 1(2):107–118, 1988.

[CA89] David Chaum and Hans Van Antwerpen. Undeniable signatures. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 212–216. Springer, 1989.

[Cam98] Jan Camenisch. *Group signature schemes and payment systems based on the discrete logarithm problem*. PhD thesis, ETH Zurich, Zürich, Switzerland, 1998.

[CBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE S&P*, 2015.

[CC18] Pyrros Chaidos and Geoffroy Couteau. Efficient designated-verifier non-interactive zero-knowledge proofs of knowledge. In *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 193–221. Springer, 2018.

[CCD+17] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE EuroS&P*, 2017.

[CCDW20] Weikeng Chen, Alessandro Chiesa, Emma Dauterman, and Nicholas P. Ward. Reducing participation costs via incremental verification for ledger systems. *IACR Cryptol. ePrint Arch.*, 2020:1522, 2020.

[CDGM19] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, pages 1639–1656. ACM, 2019.

[CDNO97] Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 1997.

[CDR14] Vincent Cheval, Stéphanie Delaune, and Mark Ryan. Tests for establishing security properties. In *TGC*, volume 8902 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2014.

[CF10] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *CCS*, 2010.

[CF13] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013.

[CFH+22] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In *CCS*, pages 455–469. ACM, 2022.

[CFN94] Benny Chor, Amos Fiat, and Moni Naor. Tracing traitors. In *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 1994.

[CGJ+17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *CCS*, pages 719–728. ACM, 2017.

[Cha90] David Chaum. Zero-knowledge undeniable signatures. In *EUROCRYPT*, volume 473 of *Lecture Notes in Computer Science*, pages 458–464. Springer, 1990.

[CHI+21] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. In *IEEE Symposium on Security and Privacy*, pages 590–607. IEEE, 2021.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In *EUROCRYPT (1)*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.

[CL06] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *CRYPTO*, 2006.

[CMS96] Jan Camenisch, Ueli M. Maurer, and Markus Stadler. Digital payment systems with passive anonymity-revoking trustees. In *ESORICS*, volume 1146 of *Lecture Notes in Computer Science*, pages 33–43. Springer, 1996.

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In *CCS*, 2014.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *CCS*, 2020.

[CvH91] David Chaum and Eugène van Heyst. Group signatures. In *EUROCRYPT*, 1991.

[CW09] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334. USENIX Association, 2009.

[Dam91] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 445–456. Springer, 1991.

[DFN06] Ivan Damgård, Nelly Fazio, and Antonio Nicolosi. Non-interactive zero-knowledge from homomorphic encryption. In *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2006.

[DGRW18] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 155–186. Springer, 2018.

[DGS+18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018.

[DHM+20] Ivan Damgård, Helene Haagh, Rebekah Mercer, Anca Nitulescu, Claudio Orlandi, and Sophia Yakoubov. Stronger security and constructions of multi-designated verifier signatures. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 229–260. Springer, 2020.

[DKPW12] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak, and Daniel Wichs. Message authentication, revisited. In *EUROCRYPT*, 2012.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[DSB+16] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual key-fingerprint representations. In *USENIX Security Symposium*, pages 193–208. USENIX Association, 2016.

[EMBB17] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate transparency with privacy. *Proc. Priv. Enhancing Technol.*, 2017(4):329–344, 2017.

[Fac17] Facebook. Messenger secret conversations technical whitepaper, 2017. `https://about.fb.com/wp-content/uploads/2016/07/messenger-secret-conversations-technical-whitepaper.pdf`.

[FDP+14] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. Hey, NSA: stay away from my market! future proofing app markets against powerful attackers. In *CCS*, pages 1143–1155. ACM, 2014.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.

[FN14] Mary Flanagan and Helen Nissenbaum. *Values at play in digital games*. MIT Press, 2014.

[FPS20] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. Blind schnorr signatures and signed elgamal encryption in the algebraic group model. In *EUROCRYPT*, 2020.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, 1986.

[FS90] Uriel Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *STOC*, pages 416–426. ACM, 1990.

[FTY96] Yair Frankel, Yiannis Tsiounis, and Moti Yung. "indirect discourse proof": Achieving efficient fair off-line e-cash. In *ASIACRYPT*, volume 1163 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1996.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR Cryptol. ePrint Arch.*, 2019:458, 2019.

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 66–97. Springer, 2017.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *CRYPTO*, 2017.

[Gro07] Jens Groth. Fully anonymous group signatures without random oracles. In *ASIACRYPT*, 2007.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2010.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[HG11] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *IEEE S&P*, 2011.

[HG13] Ryan Henry and Ian Goldberg. Batch proofs of partial knowledge. In *ACNS*, 2013.

[HHK+21] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkleˆ2: A low-latency transparency log system. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2021.

[HNC+22] Yiqing Hua, Armin Namavari, Kaishuo Cheng, Mor Naaman, and Thomas Ristenpart. Increasing adversarial uncertainty to scale private similarity testing. In *USENIX Security Symposium*, pages 1777–1794. USENIX Association, 2022.

[HYWS11] Qiong Huang, Guomin Yang, Duncan S. Wong, and Willy Susilo. Efficient strong designated verifier signature schemes without random oracle or with non-delegatability. *Int. J. Inf. Sec.*, 10(6):373–385, 2011.

[IAV22] Rawane Issa, Nicolas Alhaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender. In *USENIX Security Symposium*, pages 2335–2352. USENIX Association, 2022.

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, 2014.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, 2016.

[JSI96] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. Designated verifier proofs and their applications. In *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 143–154. Springer, 1996.

[KCDF17] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *SOSP*, 2017.

[KHP+13] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil D. Gligor. Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In *WWW*, pages 679–690. International World Wide Web Conferences Steering Committee / ACM, 2013.

[KLD20] Albert Kwon, David Lu, and Srinivas Devadas. XRD: scalable messaging system with cryptographic privacy. In *NSDI*, pages 759–776. USENIX Association, 2020.

[KM21] Anunay Kulshrestha and Jonathan R. Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In *USENIX Security Symposium*, pages 893–910. USENIX Association, 2021.

[KP05] Caroline Kudla and Kenneth G. Paterson. Non-interactive designated verifier proofs and undeniable signatures. In *IMACC*, volume 3796 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2005.

[KPS18] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 944–961. IEEE Computer Society, 2018.

[KV02] Dennis Kügler and Holger Vogt. Offline payments with auditable tracing. In *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 269–281. Springer, 2002.

[Lan16] Adam Langley. Pond, 2016. `https://github.com/agl/pond`.

[Lau14] Ben Laurie. Certificate transparency. *Commun. ACM*, 57(10):40–46, 2014.

[LdV17] Isis Agora Lovecruft and Henry de Valence. HYPHAE: Social secret sharing, 2017. `https://patternsinthevoid.net/hyphae/hyphae.pdf`.

[LGG+22] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. Aardvark: An asynchronous authenticated dictionary with applications to account-based cryptocurrencies. In *USENIX Security Symposium*, pages 4237–4254. USENIX Association, 2022.

[LGZ18] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *OSDI*, 2018.

[LKMS04] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis E. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136. USENIX Association, 2004.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *RFC*, 6962:1–27, 2013.

[LM19] Russell W. F. Lai and Giulio Malavolta. Subvector commitments with application to succinct arguments. In *CRYPTO (1)*, volume 11692 of *Lecture Notes in Computer Science*, pages 530–560. Springer, 2019.

[LNS20] Jonathan Lee, Kirill Nikitin, and Srinath T. V. Setty. Replicated state machines without replicated execution. In *IEEE Symposium on Security and Privacy*, pages 119–134. IEEE, 2020.

[LP16] Adam Langley and Trevor Perrin. Replacing group signatures with HMAC in Pond, 2016. `https://moderncrypto.org/mail-archive/messaging/2014/000409.html`.

[LPY15] Benoît Libert, Thomas Peters, and Moti Yung. Short group signatures via structure-preserving signatures: Standard model security from simple assumptions. In *CRYPTO*, 2015.

[LRTY22] Linsheng Liu, Daniel S. Roche, Austin Theriault, and Arkady Yerukhimovich. Fighting fake news in encrypted messaging with the fuzzy anonymous complaint tally system (FACTS). In *NDSS*. The Internet Society, 2022.

[Lun17] Joshua Lund. Technology preview: Sealed sender for Signal, 2017. `https://signal.org/blog/sealed-sender/`.

[LV04a] Fabien Laguillaumie and Damien Vergnaud. Designated verifier signatures: Anonymity and efficient construction from any bilinear map. In *SCN*, volume 3352 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2004.

[LV04b] Fabien Laguillaumie and Damien Vergnaud. Multi-designated verifiers signatures. In *ICICS*, volume 3269 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2004.

[LWB05] Helger Lipmaa, Guilin Wang, and Feng Bao. Designated verifier signature schemes: Attacks, new security notions and a new construction. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 459–471. Springer, 2005.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *CCS*, 2019.

[LZH+23] Junzuo Lai, Gongxian Zeng, Zhengan Huang, Siu Ming Yiu, Xin Mu, and Jian Weng. Asymmetric group message franking: Definitions and constructions. In *EUROCRYPT (5)*, volume 14008 of *Lecture Notes in Computer Science*, pages 67–97. Springer, 2023.

[MB02] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *USENIX Security Symposium*, pages 297–312. USENIX, 2002.

[MBB+15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, 2015.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[MKA+21] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. Improving Signal's sealed sender. In *NDSS*, 2021.

[MKL+20] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures. *CoRR*, abs/2011.04551, 2020.

[Nis01] Helen Nissenbaum. Securing trust online: Wisdom or oxymoron. *BUL Rev.*, 81:635, 2001.

[Nis04] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[Nis17] Helen Nissenbaum. Protecting privacy in an information age: The problem of privacy in public. *Law and Philosophy*, 2017.

[NKJ+17] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In *USENIX Security Symposium*, pages 1271–1287. USENIX Association, 2017.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security Symposium*, pages 2075–2092. USENIX Association, 2020.

[PEB21] Charlotte Peale, Saba Eskandarian, and Dan Boneh. Secure complaint-enabled source-tracking for encrypted messaging. In *CCS*, pages 1484–1506. ACM, 2021.

[PH23] Alistair Pattison and Nicholas Hopper. Poster: Committee moderation on encrypted messaging platforms. In *IEEE Symposium on Security and Privacy*. IEEE, 2023.

[PHE+17] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *USENIX Security*, 2017.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. `https://signal.org/docs/specifications/doubleratchet/`.

[PP15] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *ESORICS (2)*, volume 9327 of *Lecture Notes in Computer Science*, pages 622–641. Springer, 2015.

[RMA+23] Nathan Reitinger, Nathan Malkin, Omer Akgul, Michelle L. Mazurek, and Ian Miers. Is cryptographic deniability sufficient? Non-expert perceptions of deniability in secure messaging. In *IEEE Symposium on Security and Privacy*, pages 947–964. IEEE, 2023.

[RMM22] Michael Rosenberg, Mary Maller, and Ian Miers. Snarkblock: Federated anonymous blocklisting from hidden common input aggregate proofs. In *IEEE Symposium on Security and Privacy*, pages 948–965. IEEE, 2022.

[Rya14] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS*. The Internet Society, 2014.

[SAGL18] Srinath T. V. Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, pages 339–356. USENIX Association, 2018.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *CRYPTO (3)*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020.

[SHT21] Kavya Sreedhar, Mark Horowitz, and Christopher Torng. A fast large-integer extended gcd algorithm and hardware design for verifiable delay functions and modular inversion. *IACR Cryptology ePrint Archive*, 2021/1292, 2021.

[SKM03] Shahrokh Saeednia, Steve Kremer, and Olivier Markowitch. An efficient strong designated verifier signature scheme. In *ICISC*, volume 2971 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2003.

[SKM23] Sarah Scheffler, Anunay Kulshrestha, and Jonathan R. Mayer. Public verification for private hash matching. In *IEEE Symposium on Security and Privacy*. IEEE, 2023.

[Sno01] Alex C. Snoeren. Hash-based IP traceback. In *SIGCOMM*, pages 3–14. ACM, 2001.

[SP01] Dawn Xiaodong Song and Adrian Perrig. Advanced and authenticated marking schemes for IP traceback. In *INFOCOM*, pages 878–886. IEEE Comptuer Society, 2001.

[STV+16] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities "honest or bust" with decentralized witness cosigning. In *IEEE Symposium on Security and Privacy*, pages 526–545. IEEE Computer Society, 2016.

[SWKA00] Stefan Savage, David Wetherall, Anna R. Karlin, and Thomas E. Anderson. Practical network support for IP traceback. In *SIGCOMM*, pages 295–306. ACM, 2000.

[TAKS07] Patrick P. Tsang, Man Ho Au, Apu Kapadia, and Sean W. Smith. Blacklistable anonymous credentials: blocking misbehaving users without ttps. In *CCS*, 2007.

[TBB+17] Joshua Tan, Lujo Bauer, Joseph Bonneau, Lorrie Faith Cranor, Jeremy Thomas, and Blase Ur. Can unicorns help users compare crypto key fingerprints? In *CHI*, pages 3787–3798. ACM, 2017.

[TBP+19] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *CCS*, pages 1299–1316. ACM, 2019.

[TD17] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symposium on Security and Privacy*, pages 393–409. IEEE Computer Society, 2017.

[TFZ+21] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable registries with efficient client audits from rsa authenticated dictionaries. *IACR Cryptol. ePrint Arch.*, page 627, 2021.

[TFZ+22] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. VeRSA: Verifiable registries with efficient client audits from RSA authenticated dictionaries. In *ACM CCS*, 2022.

[TGL+17] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *SOSP*, pages 423–440. ACM, 2017.

[TGL+19a] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In *CRYPTO*, 2019.

[TGL+19b] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. *IACR Cryptol. ePrint Arch.*, page 565, 2019.

[TKCS11] Patrick P. Tsang, Apu Kapadia, Cory Cornelius, and Sean W. Smith. Nymble: Blocking misbehaving users in anonymizing networks. *IEEE Trans. Dependable Sec. Comput.*, 2011.

[TKPS22] Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath T. V. Setty. Transparency dictionaries with succinct proofs of correct operation. In *NDSS*. The Internet Society, 2022.

[TLMR21] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. Orca: Blocklisting in sender-anonymous messaging. *IACR Cryptol. ePrint Arch.*, page 1380, 2021.

[TLMR22] Nirvan Tyagi, Julia Len, Ian Miers, and Thomas Ristenpart. Orca: Blocklisting in sender-anonymous messaging. In *USENIX Security Symposium*, 2022.

[TMR19a] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In *ACM CCS*, 2019.

[TMR19b] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. *IACR Cryptol. ePrint Arch.*, page 981, 2019.

[TMS+23] Kurt Thomas, Sarah Meiklejohn, Michael A. Specter, Xiang Wang, Xavier Llorà, Stephan Somogyi, and David Kleidermacher. Robust, privacy-preserving, transparent, and auditable on-device blocklisting. *CoRR*, abs/2304.02810, 2023.

[TXN20] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis)aggregation. *IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[TY98] Yiannis Tsiounis and Moti Yung. On the security of elgamal based encryption. In *PKC*, 1998.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[vdHLZZ15] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP*, pages 137–152. ACM, 2015.

[VWO+17] Elham Vaziripour, Justin Wu, Mark O'Neill, Jordan Whitehead, Scott Heidbrink, Kent E. Seamons, and Daniel Zappala. Is that you, alice? A usability study of the authentication ceremony of secure messaging applications. In *SOUPS*, pages 29–47. USENIX Association, 2017.

[WCFJ12] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *OSDI*, pages 179–182. USENIX Association, 2012.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2019.

[WGH+] Barry Whitehat, Alex Gluchowski, HarryR, Yondon Fu, and Philippe Castonguay. Roll up / roll back snark. `https://ethresear.ch/t/roll-up-roll-back-snark-side-chain-17000-tps/`.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security Symposium*, pages 675–692. USENIX Association, 2018.

[YRC15] Jiangshan Yu, Mark Ryan, and Cas Cremers. How to detect unauthorised usage of a key. *IACR Cryptol. ePrint Arch.*, page 486, 2015.

[ZWZ+21] Ye Zhang, Shuo Wang, Xian Zhang, Jiangbin Dong, Xingzhong Mao, Fan Long, Cong Wang, Dong Zhou, Mingyu Gao, and Guangyu Sun. Pipezk: Accelerating zero-knowledge proof with a pipelined architecture. In *ISCA*, pages 416–428. IEEE, 2021.