How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)

by

Ioan Alin Tomescu Nicolescu

B.S., Stony Brook University (2012)
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author .....
Department of Electrical Engineering and Computer Science
January 28, 2020

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

How to Keep a Secret and Share a Public Key (Using Polynomial Commitments)

by

Ioan Alin Tomescu Nicolescu

Submitted to the Department of Electrical Engineering and Computer Science
on January 28, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Despite 40+ years of amazing progress, cryptography is constantly plagued by two simple problems: keeping secret keys *secret* and making public keys *public*. For example, public-key encryption is secure only if each user (1) keeps his secret key out of the hands of the adversary and (2) correctly distributes his public key to all other users. This thesis seeks to address these two fundamental problems.

First, we introduce communication-efficient, fully-untrusted append-only logs, which can be used to correctly distribute public keys. Our constructions have logarithmic-sized proofs for the two key operations in append-only logs: looking up public keys and verifying the log remained append-only. In contrast, previous logs either have linear-sized proofs or need extra trust assumptions. Our logs can also be used to secure software distribution and, we hope, to increase transparency in any institution that wants to do so.

Second, we speed up threshold cryptosystems, which protect secret keys by splitting them up across many users. We introduce threshold signatures, verifiable secret sharing and distributed key generation protocols that can scale to millions of users. Our protocols drastically reduce execution time, anywhere from 2× to 4500×, depending on the scale. For example, at large scales, we reduce time from tens of hours to tens of seconds.

At the core of most of our contributions lie new techniques for computing evaluation proofs in constant-sized polynomial commitments. Specifically, we show how to decrease the time to compute n proofs for a degree-bound n polynomial from $O(n^2)$ to $O(n \log n)$, at the cost of increasing proof size from $O(1)$ to $O(\log n)$. Our techniques could be of independent interest, as they give rise to other cryptographic schemes, such as Vector Commitments (VCs).

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor of Electrical Engineering
and Computer Science

"Kindly let me help you or you'll drown,
said the monkey, putting the fish safely up a tree."

ALAN WATTS

Acknowledgments

There are so many people I want to thank, it would be understandable if you get tired of reading this...

My advisor, **Srini Devadas**, for supporting and guiding me during these years. For the freedom to explore any topic I was interested in. For his guidance when I did not know what to do with that freedom. For trusting me to mentor four high school students. For all the pool parties. For all the ice cream trips to Ghirardelli's!

My thesis committee, **Ron Rivest** and **Vinod Vaikuntanathan**, for their invaluable feedback.

My undergraduate advisor, **Radu Sion**, for the time and energy he put into mentoring me. For his constant encouragement. For his occasionally harsh reality checks.

My research group at MIT. For inspiring and encouraging me along the way. For being the most unlikely combination of characters. **Ling Ren**, for all the research discussions in our office. **Victor Costan**, for reminding me not to stress out and have some fun. (And, for sometimes forcing me to.) **Christopher Fletcher**, for once mentioning in passing how awesome polynomials are. (Hence, this thesis.) **Quan Nguyen**, for showing us how to build CPUs in Minecraft. **Sabrina Neuman**, for teaching us about robots. **Xiangyao Yu**, for giving the lab's newest grad student a computer to work on. **Hsin-Jung Yang**, for all the FPGA talks. **Hanshen Xiao**, for being the math wizard one can always rely on. **Charles Herder**, for being the wizard of all wizards. **Albert Kwon**, for inspiring me to do work as awesome as his. **Jun Wan**, for all the breakfasts at Darwin's and for always checking my math. **Kyle Hogan**, for teaching me how to "coffee" right. **Jules Drean**, for his awesome energy. **Sacha Servan-Schreiber**, for walking the Appalachian trail and telling us about it. **Alex (Yu) Xia**, for all the times he popped up into our office to check how we're doing. **Zack Newman**, for all the cool crypto talks and for his crazy-good life habits. **Ilia Lebedev**, for being a generally questionable individual. For all those times he filled in as a TA for me in Spring 2014. For all the cocktails he made at Srini's pool parties. For being crazy enough to buy his own boat. Twice. For all the sailing trips. (Ilusha, I'm sorry I never got to go on the new boat.) For all the crazy parties together. For all of his "keynote talks" about Sable Island. For the 6 years of sharing an office that was never boring, sometimes to the detriment of our research goals. (Really, Ilusha deserves his own section.)

Sally O. Lee, for all the jokes and laughter. For having the most hilarious office wall. For lending me her mother's lovely, Christmas-themed coffee mug, which made all of my mornings in the Stata Center feel so cozy.

My four high school students from the MIT PRIMES program, **Vivek Bhupatiraju**, **Robert Chen**, **John Kuszmaul** and **Yiming Zheng**, for the joy of creating and solving problems together. This thesis is based on joint work with them.

My other collaborators, for making all of this work possible. **Peter Williams**, for showing me what an "Aha!" moment looks like while he worked out the file system design for PrivateFS. For trusting me to implement this design. **Mashael AlSabah**, for the fun discussions and collaboration around email security. **Nikos Triandopoulos**, for opening up his door to me when I was a lost graduate student. For the wonderful collaboration that ensued. **Dimitris Papadopoulos**, for all the late phone calls discussing research. For his

constant encouragement and his friendship. **Babis Papamanthou**, for getting as excited about the append-only dictionary problem as me (if not more) and his constant efforts to help improve and position the result.

My collaborators at VMware Research Group (VRG). **Dahlia Malkhi**, my first VRG mentor. For all the fun-filled, three-hour meetings we had during the summer of 2017. For bringing an incredibly-positive, highly-contagious energy to every meeting. For her invaluable guidance. For an incredibly productive summer that re-instilled confidence in me. **Ittai Abraham**, my second VRG mentor, for our wonderful remote collaboration. For all those long phone calls about blind signatures, threshold signatures and other fantastic beasts. **Mike Reiter**[1], for his support when I needed it. The only thing better than a meeting with Dahlia was a meeting with Mike and Dahlia. **Benny Pinkas**, for all the helpful discussions about RSA threshold signatures, polynomial interpolation and Fast Fourier Transforms. **Guy Golan-Gueta**, for always being open to me changing his existing PBFT codebase. For all the long talks on speeding up threshold signatures in SBFT. **Soumya Basu** and **Adi Seredinschi**, for all the productive conversations while at VMware and all the fun times while outside it.

All of my fellow interns from VRG, with whom I spent two fantastic summers in California.

The many folks that have helped and steered me along the way. **Marten van Dijk**, for all the interesting discussions inside and outside our group meetings. **Neha Narula**, for all the discussions about proof-of-work consensus. **Madars Virza**, for our conversations on using polynomials to solve the append-only dictionary problem. **Henry-Corrigan Gibbs**, for the conversation about accumulators and append-only dictionaries. **Nickolai Zeldovich**, for letting me pick his brain on append-only dictionaries and for his advice on postdocs. **Michael Alan Specter**, for all our conversations on applied crypto research.

Nancy Lynch, my academic advisor, whom I would meet with every semester to discuss my progress (or lack thereof). I was always a bit nervous meeting with her "empty handed" in my first years. Yet she always emanated a silent, reassuring confidence that I would find my way.

My family, for a loving home.

My mother, for all her sacrifices. For raising us. For letting us go. For never doubting us. For teaching us independence and responsibility through her faith in us. For being a role model. For her boundless love.

My brother, for crawling, walking and running through this life with me. For letting me spend more time than him on the computer.

My sister, for loving us the moment she met us. For being an inspiration (though she would probably laugh at this notion).

My father. For his existence. For giving me the rare opportunity to study in the United States. For all his encouragement.

My step-mother. For moving to Romania. For teaching me how to write. For taking care of my father.

Naşu, for being a father figure.

[1]The author shall not acknowledge Mike Reiter's controversial victory in the VRG Chin-up Olympics of 2017, which was indubitably attributed to improper chin-up technique. Video evidence can be provided upon request.

Naș, for letting me keeping the TV on max volume so I could hear the English being spoken. For her brilliant sense of humor.

Mela, for teaching me English. For her faith in us. For her limitless love.

My friends from MIT, for keeping me sane. *Ariel Anders*, *Alex Băcanu*, *Twan Koolen*, *Marek Hempel*, *Danielle Pace*, *"Bald" Mike*, *Colm O'Rourke*, *Gautam Kamath*, *Sam Park*, *Julius Adebayo*, *Eva Golos*, *Sara Achour*, *Rohit Singh*, *Michel "Mon Amour..." Babany*, *Frank Permenter*, *Orhan Celiker*, *Jessica Ray*, *Ognyan Georgiev*, *Andre Urgiles*, *Fernando Couto*, *Cesar Cruz*, *Veronica Mirta*, *Avelino Tolentino*, *Affi Maragh*, *Ina Kundu*, *Po-An (Ben) Tsai*, *Chetan Manikantan*, *David Qiu*, *Garrett Dowdy*, *João Ramos*, *Lindsay Brownell*, *Mark Jeffrey*, *Pavel Chykov*.

The Romanian Mafia, at MIT and beyond, *Sorin Grama*, *Florin Chelaru*, *Radu Berinde*, *Răzvan "Tati!" Marinescu*, *Iulia Mădălina Streangă*, *Suzana Iacob*, *Liliana Oniț-Lenco*, *Elena Solomon*, *Andreea Bobu*, *Andreea Bodnari*, *Dana Jinaru*, *Daniel Farmache*, *Livia Dinu*, *Ovidiu Tisler*, *Teo Cucu*, *Victor Pankratius*, *Alin Dragoș*.

My gym bros, *Sam "Teach me how to deadlift" Park*, *Ivan "It's spinal!" Kuraj*, *James Noraky*, *Jun Wan* and *Colm O'Rourke*, for all the fun times at the Z Center and Stata Center gyms.

My friends from college, *Sharif Syed*, *Brian Wilson*, *Prashant Makwana*, *Ezra Margono*, *Mihai Albu*, *Radu Zamfir*, *Mike Boruta*, *Luke Mladek*, *Jan Kasiak*, *Farhad Zaman*.

My friends from Romania, *Alex "Titisan" Tiutiu*, *Ana (Anița) Maria*, *Ioana Bălan*, *Maria Popescu*, *Badea "Mutu" Mihai*, *Cătălin "Cole" Apostol*, *Liviu "roacăru" Nicolae*, *Cristi "Titus" Niță*, *Silviu "But" Niculae*, *George "Hesus" Postelnicu*, *Dana Velea*, *Adrian Plăcintescu*, *Bobi "Lion Heart" Niță*.

My physical therapist, **Dr. Lucia Hamilton**, who helped me fix my scapular dysfunction issue. For putting up with me as her patient. For teaching me not to over-think movement. For the gift of "lifting things up and putting them back down again".

To everyone who has ever been a part of my life, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 How to Share a Public Key

Almost every cryptographic scheme assumes actors know each other's public keys. This assumption is often referred to as the *public key infrastructure* (PKI) assumption. For example, a PKI enables users to safely shop online on Amazon.com, to message each other securely on WhatsApp, and to use social media on LinkedIn.com. Yet, despite their absolute necessity for security, state of the art PKIs remain inadequate in several ways.

First, PKIs rely on the unreasonable assumption of *trusted parties*. For example, Facebook, who develops and owns WhatsApp, runs a *public key directory* (PKD) mapping each user's phone number to that user's public key. This *centralized* control of the PKD by Facebook is quite natural. After all, the PKD is a crucial component of WhatsApp's infrastructure and needs incentives to be maintained, incentives which Facebook has. Unfortunately, this centralized approach assumes we live in a world where Facebook cannot be compromised or coerced. In reality, mass surveillance revelations tell us the exact opposite story [2, 103]. Thus, it is not unfathomable for a rogue Facebook to *man-in-the-middle* "secure" WhatsApp conversations. Indeed, this would not be too difficult either. To snoop on two users, say *Alice* and *Bob*, Facebook need only *equivocate* about their public keys in the PKD.

Second, when trusted parties go rogue, *PKI attacks cannot be easily detected by victims* like Alice and Bob. We focus on "detection" rather than "prevention," since it is impossible to prevent PKI attacks in this strong, pessimistic model where trusted parties can go rogue. Fortunately, post-attack detection is still possible, although not always easily achieved, as we detail next.

Consider the following (rather contrived) example. Alice and Bob's WhatsApp conversation is man-in-the-middled by Facebook. Fortunately, they both "record" all of their encrypted WhatsApp traffic. They then meet in a coffee shop, inspect this traffic and notice that, at some point, their public keys were changed to unrecognized *fake* ones that were subsequently used to encrypt their messages. They then conclude a man-in-the-middle attack occurred. However, their effort is considerable and does not scale to detecting attacks in multiple conversations. Thus, more efficient detection methods are needed.

Third, (some) PKIs suffer from a *weakest link in the chain* problem. In other words, while one trusted party is an unreasonable assumption, $n > 1$ trusted parties is much worse, since

it increases the attack surface. Sadly, the PKI used to secure HTTPS connections is the best example of this. The *Web PKI* is comprised of 1400+ *Certificate Authorities* (CAs) [81] which digitally sign *certificates* that map a website to its public key. Quite terrifyingly, *any* CA, in *any* country, can sign a certificate for *any* website. Not surprisingly, this has led to many websites being surreptitiously *impersonated* [145, 161]. Even worse, it is conceivable that some impersonation attacks went undetected. Fortunately, in 2013, Google took a key step towards *detecting* and thus *deterring* such attacks [137], which we discuss next.

Maintaining Trust in PKIs with Transparency Logs. In 2011, Google was the victim of two impersonation attacks [4, 80]. As a result, in March 2013, Google deployed *Certificate Transparency* (CT) [137] as a way of detecting impersonation attacks and, hopefully, deterring them by publicly (though not provably) revealing their traces. The key idea behind CT is to *require* that all certificates for websites be published in a *transparency log*. In other words, unless a certificate comes with an *inclusion proof* in the transparency log, a browser such as Firefox will never accept that certificate. This way, if a rogue CA issues a fake certificate for a victim website, then the victim can detect this attack by searching or *monitoring* in the transparency log. This immediately raises the question of who runs this log and what if, just like a CA, it misbehaves? The answer is an untrusted *log server* runs the log but, if it misbehaves, it can be caught either (1) by *domain owners* such as `google.com`, `yahoo.com` or `your-website.com`, or (2) by users like Alice and Bob, or (3) by any other third-party *verifier*. Importantly, this misbehavior can be *cryptographically proven*.

It is important to understand that CT logs cannot *prevent* impersonation attacks, since a rogue CA can always ("kamikaze-style") publish a fake certificate for, say, `google.com` in the log which will be used by (unsuspecting) users, until the fake certificate is detected and revoked by Google. Instead, CT logs make such attacks quickly *detectable* by the impersonated victim. Specifically, the victim can now search the log for its certificates (i.e., monitor) and discover the fake certificate. Unfortunately, this does not prove the certificate is "fake." After all, the CA can falsely claim the victim was the one who asked for this certificate to be issued. Since there is no cryptographic way of figuring out the truth, this must be resolved in the real world via the judicial system. This way, incompetent or malicious CAs can be eliminated from the Web PKI, leading to a more trustworthy ecosystem. Furthermore, the threat of legal action should deter CAs from misbehaving. Yet this is only possible if logs are honest, which brings us to the focus of Part I of this thesis: *cryptography for transparency logs.*

Transparency Logs with Succinct Proofs. To operate correctly, transparency logs must have three key properties. First, when a browser connects to a website and receives a certificate, the log server should prove to the browser that the certificate is included in the log via an *inclusion proof*. Otherwise, instead of being forced to publish fake certificates in the log and risk detection, rogue CAs can directly (and thus undetectably) present fake certificates to victim browsers.

Second, when a domain owner *looks up* their own certificates in the log, the server should prove that the domain's complete set of certificates has been returned via a *lookup proof*. Otherwise, if a fake certificate can be left out of the answer, then it would never be discovered by victims monitoring the log, leading to undetected impersonation attacks.

Third, logs should prove to everybody that they remain append-only via *append-only proofs*. Otherwise, a fake certificate for a victim website can be inserted in the log, shown to a victim browser and removed quickly, before the victim website has a chance to monitor its certificates.

Unfortunately, although all transparency log designs support these proofs, the proof sizes are too large [137, 147, 183]: either the lookup proof or the append-only proof is linear in the size of the log. As a workaround to obtain small proof sizes, all transparency logs introduce additional trust assumptions or additional actors. Specifically, they either introduce additional *verifiers* who check the log for misbehavior on behalf of users [137, 183], or rely on non-collusion assumptions [16, 131, 199], or require users to *frequently check the log* [147, 183, 213]. As a result, such constructions are either difficult to deploy [16, 131, 199], or expensive in terms of communication overhead [137, 147], or slow to detect attacks [183, 213].

In contrast, logs with succinct proofs do not suffer from these problems. First, they require no additional verifiers, making them easier to deploy. Second, by definition, they have low communication overhead. Third, since proof sizes are small, they can be frequently queried to detect attacks fast. Thus, the first question this thesis asks is:

Question 1: *Can we build transparency logs with succinct lookup proofs and succinct append-only proofs in the single, untrusted sever model?*

We answer this question positively in Part I. We give constructions for logs that map application-specific *keys* to one or more application-specific *values*, in an append-only fashion. Such logs are ideal for building public-key directories (i.e., a "key" corresponds to a domain and the "values" correspond to the certificates of that domain). We also give constructions for logs that maintain an append-only set of application-specific *elements*. Such logs are ideal for keeping track of revoked certificates [136]. Our transparency logs are useful not just for securing PKIs, but also for securing software distribution [7, 86, 119, 162, 197, 200]. Although our constructions offer small proof sizes, they are not yet fast enough to append to, at least not at the scale of a realistic PKI. We believe this can be addressed by future work and we discuss possible directions in Section 11.1.

1.2 How to Share a Secret (With Two Million People)

Many cryptographic schemes require that some information be kept secret from the adversary. For example, in public-key encryption schemes [78, 182], Alice and Bob have to keep their *private keys* secret, or else the adversary can snoop on their conversation. Similarly, in cryptocurrencies like Bitcoin [155], users have to keep their private keys secret, or else the adversary can steal their coins. Yet this requirement of keeping a private key secret from the adversary is in constant tension with the need to use it (and thus temporarily expose it) for achieving cryptographic goals: e.g., sending an encrypted message, or transferring coins. Fortunately, one way to address this tension is via *threshold cryptosystems*.

In a threshold cryptosystem, the private key is *split up* amongst a set of n players, such that *only* subsets of size $\geq t$ players can collaborate to use the key. Threshold cryptosystems better hide the key from the adversary, who now has to compromise at least t players. At the same time, they preserve functionality by allowing the players to collaborate to use the key. Importantly, players can use the key *without ever exposing it* [73–75].

Threshold Signatures. For example, a digital signature scheme such as RSA [182] or BLS [34] can be transformed into a *threshold signature scheme* (TSS) by splitting up its private key amongst n *signers*. This way, each signer has a *share* of the private key and only subsets of $\geq t$ signers can collaborate to produce a signature. Importantly, the signers can do this *without* reconstructing the private key "in plain sight" and thus exposing it.

In a TSS, signing a message m is an interactive protocol. Each signer is given m and computes a *signature share*, which is just a signature on m but under that signer's share of the private key. Then, any $\geq t$ signers get together and "aggregate" their signature shares into the *final signature* on m. As a result, the private key is never exposed but can still be used to sign!

TSS schemes have many important applications. First, they can be used to decentralize Certificate Authorities (CAs) in PKIs (see Section 1.1) by distributing their extremely-sensitive private key amongst many sub-CAs [197]. Second, they can be used to create very simple but resilient *randomness beacons* [52, 130, 198], which have many applications in cryptography [38]. Third, they can be used to create simple voting protocols and thus to scale consensus protocols [105].

While applications such as decentralizing Certificate Authorities (CA) do not require a high number of signers, other applications such as voting could require millions of signers. Unfortunately, naively-implemented TSS schemes do not scale to such a large number of signers. This is because all current TSS scheme implementations use a naive polynomial interpolation algorithm [24], which runs in time quadratic in the number of signers n. Thus, the second question this thesis asks is:

Question 2: *Can we scale threshold signature schemes to millions of signers?*

We answer this question positively in Section 10.1.1. We adapt existing, quasilinear-time interpolation algorithms [207] to work with discrete log-based signature schemes such as BLS [34]. This gives us a dramatic speed-up over naive, quadratic-time interpolation algorithms. Specifically, our threshold BLS signature [28] can aggregate a signature from one million signers in 46 seconds, a drastic improvement over 1.5 days, if done using a naive, quadratic-time interpolation algorithm.

Bootstrapping Threshold Signatures. Yet a key challenge still remains: even if TSS schemes scaled to millions of signers, how can such a large-scale TSS be securely bootstrapped? In other words, how will the private key be split up amongst the n signers so that nobody learns it but everybody has a share of it? If a single *trusted dealer* were to generate the private key and split it amongst the signers using *secret sharing* [61, 191], then he would be trusted to forget the private key. This dealer would thus become a single point of failure, which defeats the point of a threshold cryptosystem.

Fortunately, *distributed key generation* (DKG) protocols [99, 121, 126, 172] can remove this single point of failure when bootstrapping a TSS. In a DKG, the n signers jointly and randomly pick the private key and split it amongst each other, without any signers learning the final private key. In other words, the n signers can be used to "implement" an always-honest dealer. Unfortunately, DKG protocols can be computationally-expensive, requiring computation quadratic in the number of signers n [126]. Thus, the second question this thesis asks is:

Question 3: *Can we scale distributed key generation (DKG) protocols for discrete log-based cryptosystems to millions of players?*

We answer this question positively in Part II. First, we present a novel DKG protocol, which can generate keys for TSS schemes with millions of players in hours, rather than a year (see Section 10.4.3). In this process, we also introduce a scalable Verifiable Secret Sharing (VSS) protocol [61, 89, 191] and a novel Vector Commitment (VC) scheme (see Section 9.2). While VSS is a key component of DKG protocols, VCs are of independent interest and have applications to stateless cryptocurrencies [31, 58]. Second, we present a threshold-variant of BLS signatures [28, 34] that can aggregate a signature from one million signers in 46.26 secs, a drastic improvement over 1.59 days, if done naively.

1.3 Organization

This thesis is organized into two parts. The necessary, common background is in Chapter 2.

Part I. This part covers our append-only authenticated data structures for transparency logs [137, 183] with application to public-key distribution. In Chapter 3, we introduce transparency logs, motivate the need to build communication-efficient logs without trusted third parties and discuss related work. In Chapter 4, we give additional background on *cryptographic accumulators* [22, 160], which our data structures are built upon.

In Chapter 5, we introduce *append-only authenticated sets (AAS)*, which can be used to build *revocation transparency (RT)* logs [136]. First, in Section 5.2 we formalize AAS correctness and security definitions. Second, in Section 5.3, we construct an AAS from *bilinear accumulators* [160]. Third, in Section 5.4 we construct an AAS from *RSA accumulators* [22]. Finally, we prove these two constructions are secure in Section 5.3.6 and Section 5.4.4, respectively.

In Chapter 6, we introduce *append-only authenticated dictionaries (AADs)*, which can be used to build general-purpose transparency logs [83, 110]. First, in Section 6.2, we formalize AAD correctness and security definitions. Second, in Section 6.3, we show how to build an AAD from any cryptographic accumulator by slightly modifying our AAS constructions from Chapter 5. Third, in Section 6.4, we implement and evaluate our bilinear-based AAD construction.

Part II. This part covers our scalable threshold cryptosystems: threshold signature schemes (TSS), verifiable secret sharing (VSS) protocols and distributed key generation (DKG) protocols. In Chapter 7, we introduce threshold cryptosystems, motivate their need to scale and discuss related work. In Chapter 8, we give additional background on TSS, VSS and DKG protocols.

In Chapter 9, we introduce *authenticated multipoint evaluation trees (AMTs)*, a new way of precomputing evaluation proofs in polynomial commitments [128]. AMTs will be the key ingredient for scaling VSS and DKG protocols later on. In Section 9.2, we give a new *vector commitment (VC)* scheme based on AMTs.

In Chapter 10, we give scalable constructions for TSS, VSS and DKG protocols. First, in Section 10.1, we show how a faster algorithm for polynomial interpolation can be used to

scale threshold signatures to millions of signers. Second, in Section 10.2, we use AMTs to scale VSS protocols to millions of players. Third, in Section 10.3, we use our scalable VSS to scale DKG protocols. Finally, in Section 10.4, we implement and evaluate our scalable TSS, VSS and DKG protocols.

Future Work and Conclusion. In Section 11.1, we give directions for future work on AADs, discussing both how to improve our current constructions and how to develop new ones. In Section 11.2, we give directions for future work on threshold cryptosystems, with an emphasis on reducing communication in VSS and DKG protocols. We conclude in Chapter 12.

1.4 Publications

The results of this thesis are also published in two conference papers [201, 202]:

"Transparency Logs via Append-only Authenticated Dictionaries", Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos and Srinivas Devadas, in ACM CCS 2019

"Towards Scalable Threshold Cryptosystems", Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta and Srinivas Devadas, in IEEE S&P 2020

Chapter 2

Preliminaries

Let λ denote a security parameter for the many cryptosystems presented in this thesis. Let $\mathcal{H}$ denote a collision-resistant hash function (CRHF) with 2λ-bits output. We use multiplicative notation for all algebraic groups in this thesis. Let $\mathbb{F}_p$ denote the finite field "in the exponent" associated with a group $\mathbb{G}$ of prime order p. Let $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$ denote the finite field of integers modulo a prime p. Let $\mathbb{F}_p[X]$ denote all univariate polynomials with coefficients in $\mathbb{F}_p$. Let $1_{\mathbb{G}}$ denote the identity element of a group $\mathbb{G}$. We use $\mathbb{G} = \langle g \rangle$ to indicate g is a *generator* of the group $\mathbb{G}$. Let $s \in_R S$ denote sampling an element s uniformly at random from some set S. Let $\deg \phi$ denote the degree of a univariate polynomial ϕ. We say a polynomial ϕ has *degree-bound* m if $\deg \phi < m$. Let $\mathsf{poly}(\cdot)$ denote any function upper-bounded by some univariate polynomial. Let $\log x$ be shorthand for $\log_2 x$. Let $[n] = \{1, 2, \ldots, n\}$ and $[i, j] = \{i, i+1, \ldots, j-1, j\}$.

In this thesis, we assume group and field operations can be performed in $O(1)$ time. This means the security parameter λ will not typically show up in our complexities. For example, we say multiplying two field elements takes $O(1)$ time rather than $O(\lambda \log \lambda \log\log \lambda)$ time, which is typically the case.

2.1 Cryptographic Assumptions

2.1.1 Bilinear Pairings

Our work relies on the use of *pairings* or *bilinear maps* [123, 149]. Recall that a bilinear map $e(\cdot, \cdot)$ has useful algebraic properties: $e(g^a, g^b) = e(g^a, g)^b = e(g, g^b)^a = e(g, g)^{ab}$. For clarity, our cryptosystems are all presented using Type I pairings [94]. Nonetheless, our results can be re-stated using (the more efficient) asymmetric Type II and III pairings in a straightforward manner. In fact, all our implementations use (the more efficient) Type III asymmetric pairings. We often discuss the implications of this in our experiments.

Definition 2.1.1 (Bilinear pairing parameters)**.** Let $\mathcal{G}_{\mathsf{prime}}(\cdot)$ be a randomized, polynomial-time algorithm with input a security parameter λ. Then, $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$ are called *bilinear pairing parameters* if,

- $\mathbb{G} = \langle g \rangle$ and $\mathbb{G}_T$ are cyclic groups of prime (known) order p

- No PPT adversary can solve discrete log (DL) in $\mathbb{G}$ nor in $\mathbb{G}_T$, except with probability negligible in λ
- $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a bilinear map such that $\mathbb{G}_T = \langle e(g,g) \rangle$ (i.e., $e(g,g)$ generates $\mathbb{G}_T$).

2.1.2 ℓ-Strong (Bilinear) Diffie-Hellman

Definition 2.1.2 (ℓ-Strong Diffie-Hellman (SDH) Assumption)**.** Given as input security parameter 1^λ, bilinear pairing parameters $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$, public parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(g;\tau) = \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell} \rangle$ where $\ell = \mathsf{poly}(\lambda)$ and τ is chosen uniformly at random from $\mathbb{Z}_p^*$, no probabilistic polynomial-time adversary can output a pair $\langle c, g^{\frac{1}{\tau+c}} \rangle$ for some $c \in \mathbb{Z}_p \backslash \{-\tau\}$, except with probability negligible in λ.

Definition 2.1.3 (ℓ-Strong Bilinear Diffie-Hellman (SBDH) Assumption)**.** Given as input security parameter 1^λ, bilinear pairing parameters $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$, public parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(g;\tau) = \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell} \rangle$ where $\ell = \mathsf{poly}(\lambda)$ and τ is chosen uniformly at random from $\mathbb{Z}_p^*$, no probabilistic polynomial-time adversary can output a pair $\langle c, e(g,g)^{\frac{1}{\tau+c}} \rangle$ for some $c \in \mathbb{Z}_p \backslash \{-\tau\}$, except with probability negligible in λ.

2.1.3 ℓ-Power Knowledge of Exponent

First, let us define the public parameters associated with the ℓ-Power Knowledge of Exponent (ℓ-PKE) assumption as:

$$\mathsf{PP}_\ell^{\mathsf{PKE}}(g;\tau,\beta) = \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell}, g^{\beta\tau}, g^{\beta\tau^2}, \ldots, g^{\beta\tau^\ell} \rangle$$

Definition 2.1.4 (ℓ-Power Knowledge of Exponent (ℓ-PKE) Assumption)**.** For all probabilistic polynomial-time adversaries $\mathcal{A}$, there exists a probabilistic polynomial time extractor χ_A such that for all *benign* auxiliary inputs $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$

$$\Pr\left[\begin{array}{c} \langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda); \langle \tau, \beta \rangle \in_R \mathbb{Z}_p^*, \\ \sigma \leftarrow \langle \mathbb{G}, \mathbb{G}_T, p, g, e, \mathsf{PP}_\ell^{\mathsf{PKE}}(g;\tau,\beta) \rangle, \\ \langle c, \hat{c}; a_0, a_1, \ldots, a_\ell \rangle \leftarrow (\mathcal{A} || \chi_{\mathcal{A}})(1^\lambda, \sigma, z) : \\ \hat{c} = c^\beta \wedge c \neq g^{\prod_{i=0}^{\ell} a_i \tau^i} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

where $\langle y_1; y_2 \rangle \leftarrow (\mathcal{A} || \chi_{\mathcal{A}})(x)$ means $\mathcal{A}$ returns y_1 on input x and χ_A returns y_2 given the same input x and $\mathcal{A}$'s random tape. Auxiliary input z is required to be drawn from a benign distribution to avoid known negative results associated with knowledge-type assumptions [26, 40].

The ℓ-PKE assumption is non-standard and often referred to as "non-falsifiable" in the literature. This terminology can be confusing, since previous, so-called "non-falsifiable" assumptions have been falsified [18]. Naor explored the nuance of these types of assumptions and proposed thinking of them as "not efficiently falsifiable" [157]. For example, to falsify ℓ-PKE one must find an adversary $\mathcal{A}$ that outputs $\hat{c} = c^\beta$ and then mathematically prove that *all* extractors $\chi_{\mathcal{A}}$ fail for it.

2.2 Roots of Unity and Fast Fourier Transforms (FFTs)

We use FFT [206] to speed up many polynomial operations in $\mathbb{F}_p[X]$ where p is a prime. For this to work, we assume a finite field $\mathbb{F}_p$ which supports *roots of unity* and thus supports the discrete-version of the Fast Fourier Transform. (Cormen et al. [63] and von zur Gathen and Gerhard [206] give an excellent background on roots of unity and FFT.) Let $N = 2^k$ for some $k > 0$ and let ω_N denote a primitive Nth root of unity in the finite field $\mathbb{F}_p$. Recall that an FFT of size N on a degree-bound N polynomial ϕ computes $(\phi(\omega_N^{i-1}))_{i\in[N]}$ in $\Theta(N \log N)$ times. Also, recall that the set $\{\omega_N^0, \omega_N^1, \omega_N^2, \ldots, \omega^{N-1}\}$ of all N Nth roots of unity forms a multiplicative subgroup of $\mathbb{F}_p$.

FFT can be used to speed up polynomial multiplication and division. Specifically, for polynomials of degree-bound n, we divide and multiply them in $\Theta(n \log n)$ field operations [5, 179]. We compute Bézout coefficients γ, ζ for two polynomials α, β of degree-bound n such that $\gamma(x)\alpha(x) + \zeta(x)\beta(x) = \gcd(\alpha, \beta)$ using the Extended Euclidean Algorithm (EEA) in $\Theta(n \log^2 n)$ field operations [205].

2.3 Efficiently Evaluating Polynomials at Many Points

Part II of this thesis builds upon *polynomial multipoint evaluation* techniques [207]. Given a degree t polynomial ϕ, naively evaluating it at $n > t$ points $x_1, \ldots, x_n$ requires $\Theta(nt)$ time. This is fast when t is very small relative to n but can be slow when $t \approx n$, as is the case in many instantiations of threshold cryptosystems. Fortunately, a multipoint evaluation reduces this time to $O(n \log^2 n)$ using a divide and conquer approach. Specifically, one first computes $\phi_L(x) = \phi(x) \bmod (x - x_1)(x - x_2) \cdots (x - x_{n/2})$ and then $\phi_R(x) = \phi(x) \bmod (x - x_{n/2+1})(x - x_{n/2+2}) \cdots (x - x_n)$ Then, one simply recurses on the two *half-sized* subproblems: evaluating $\phi_L(x)$ at $x_1, x_2, \ldots, x_{n/2}$ and $\phi_R(x)$ at $x_{n/2+1}, x_{n/2+2}, \ldots x_n$. Ultimately, the leaves of this recursive computation store $\phi(x) \bmod (x - x_i)$, which is exactly $\phi(i)$ by the polynomial remainder theorem (see Figure 2-1).

For example, consider the multipoint evaluation of ϕ at $\{1, 2, \ldots, 8\}$, which we depict in Figure 2-1. We start at the root node ε. Here, we divide ϕ by the *accumulator polynomial* $(x-1)(x-2)\ldots(x-8)$ obtaining a *quotient polynomial* $q_{1,8}$ and *remainder polynomial* $r_{1,8}$. Then, its left and right children divide $r_{1,8}$ by the left and right "half" of $(x-1)(x-2)\ldots(x-8)$, respectively. This proceeds recursively: each node w divides $r_{\mathsf{parent}(w)}$ by its accumulator a_w, obtaining a quotient q_w and remainder r_w such that $r_{\mathsf{parent}(w)} = q_w a_w + r_w$. Note that all accumulator polynomials a_w can be computed in $O(n \log^2 n)$ time by starting with the $(x - i)$ monomials as leaves of a binary tree and "multiplying up the tree." Since division by a degree-bound n accumulator takes $O(n \log n)$ time, the total time is $T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$ [207].

2.4 Lagrange Polynomial Interpolation

Given d pairs $(x_i, y_i)_{i\in[d]}$, we can find (or *interpolate*) the unique, degree-bound d polynomial ϕ such that $\phi(x_i) = y_i, \forall i \in [d]$ using *Lagrange interpolation* [24]. First, we compute *Lagrange*

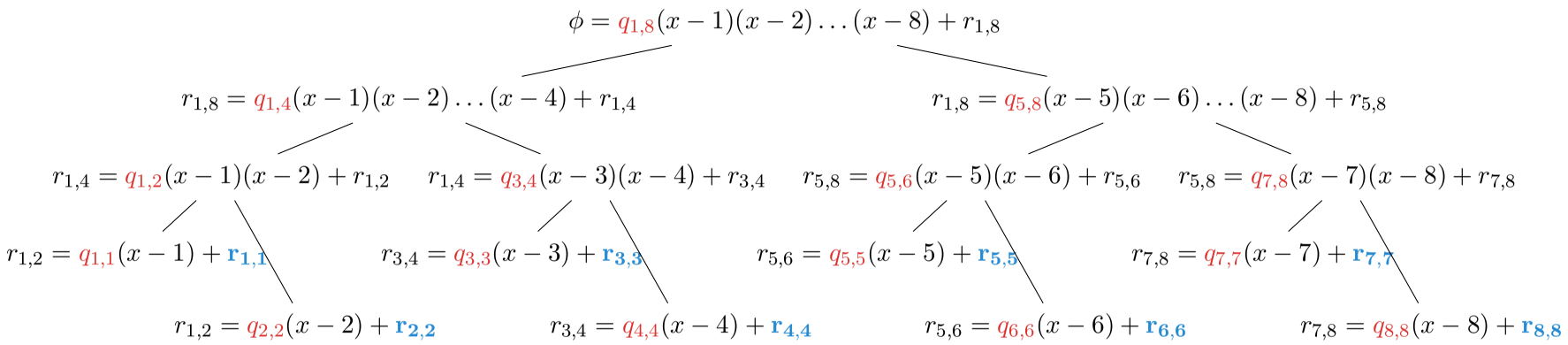


Figure 2-1: A multipoint evaluation of polynomial ϕ at points $\{1, 2, \dots, 8\}$. Each node is expressed as $a = q \cdot b + r$: i.e., a polynomial a is being divided by b, resulting in a *quotient* q and a *remainder* r. In the root node, ϕ is divided by the root *accumulator* $\prod_{i\in[8]}(x-i)$, obtaining a quotient $q_{1,8}$ and a remainder $r_{1,8}$. Then, the root's left child divides $r_{1,8}$ by $(x-1)\cdots(x-4)$ while the right child divides it by $(x-5)\cdots(x-8)$. The process is repeated recursively on the resulting $r_{1,4}$ and $r_{5,8}$ remainders. The remainders $r_{i,i}$ in the leaves are the evaluations $\phi(i)$.

polynomials $\mathcal{L}_i(x)$ for all $i \in [d]$:

$$\mathcal{L}_i(x) = \prod_{\substack{j \in [d] \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \tag{2.1}$$

Note that $\forall i \in [d], \mathcal{L}_i(x_i) = 1$ and $\forall i \in [d], \forall j \neq i, \mathcal{L}_i(x_j) = 0$. Then, we can interpolate ϕ as:

$$\phi(x) = \sum_{i \in [d]} \mathcal{L}_i(x) y_i \tag{2.2}$$

It is easy to see that $\forall i \in [d], \phi(x_i) = y_i$:

$$\begin{aligned}
\phi(x_i) &= \sum_{j \in [d]} \mathcal{L}_j(x_i) y_j \\
&= \mathcal{L}_i(x_i) y_i + \sum_{\substack{j \in [d] \\ j \neq i}} \mathcal{L}_j(x_i) y_j \\
&= 1 \cdot y_i + \sum_{\substack{j \in [d] \\ j \neq i}} 0 \cdot y_j \\
&= y_i
\end{aligned}$$

Because the interpolated polynomial is unique, this means a degree-bound d polynomial ϕ can be interpolated from *any* d evaluations $(x_i, \phi(x_i))_{i \in [d]}$. This property was used by Shamir [191] to construct secret sharing schemes (see Section 8.4) and we use it frequently in this thesis in Part II.

As described here, Lagrange interpolation can be very slow, since it requires computing all d degree-bound d Lagrange polynomials. If done naively, this takes $O(d^3)$ time. Fortunately, faster $\Theta(d \log^2 d)$ algorithms exist [207] which leverage fast polynomial multiplication, division and multipoint evaluation (see Section 2.3). In Section 10.1, we adapt one of these algorithms to speed up threshold signature schemes.

2.5 (Constant-sized) Polynomial Commitments

Most of the results in this thesis are based on *polynomial commitment schemes* [45, 89, 128, 173, 215]. In Part I, we use constant-sized polynomial commitments [128] to build various append-only authenticated data structures. In Part II, we use the same commitments to scale VSS and DKG protocols.

2.5.1 Definitions

In this section, we formally (re)define polynomial commitment schemes over $\mathbb{F}_p[X]$. Our definitions resemble the ones from previous work [128], except for small modifications.

2.5.1.1 Polynomial Commitments API

A *polynomial commitment scheme* is a tuple of six algorithms:

$\mathsf{Poly.Setup}(1^\lambda, \ell) \to \mathsf{pp}$. Generates public parameters pp for the scheme. λ is a security parameter. Some schemes require an *upper bound* ℓ on the degree of the committed polynomials. In other words, such schemes can commit only to polynomials ϕ of $\deg(\phi) \leq \ell$.

$\mathsf{Poly.Commit}(\mathsf{pp}, \phi) \to c, \delta$. Returns a commitment c to the polynomial ϕ and *decommitment information* δ. For example, for information-theoretic hiding schemes, δ will contain the randomness of the commitment. For other schemes, δ could be empty.

$\mathsf{Poly.VerifyPoly}(\mathsf{pp}, c, \phi, \delta) \to \{T, F\}$. Checks if c is a commitment to the polynomial ϕ created with decommitment information δ.

$\mathsf{Poly.ProveEval}(\mathsf{pp}, \phi, i, \delta) \to \pi$. Computes an *evaluation proof* π attesting to the correct evaluation of $\phi(x)$ at $x = i$. Some schemes will require the decommitment information δ.

$\mathsf{Poly.VerifyEval}(\mathsf{pp}, c, i, v, \pi) \to \{T, F\}$. If ϕ is the polynomial committed in c, then verifies that $\phi(i) = v$ via the evaluation proof π.

2.5.1.2 Correctness and Security

Definition 2.5.1 (Polynomial Commitment Scheme)**.** ($\mathsf{Poly.Setup}$, $\mathsf{Poly.Commit}$, $\mathsf{Poly.VerifyPoly}$, $\mathsf{Poly.ProveEval}$, $\mathsf{Poly.VerifyEval}$) is a secure *polynomial commitment scheme* with *computational hiding* if $\forall$ upper-bounds $\ell = \mathsf{poly}(\lambda)$ it satisfies the following properties:

Definition 2.5.2 (Opening Correctness)**.** $\forall \phi \in \mathbb{F}_p[X]$ with $\deg \phi \leq \ell$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Poly.Setup}(1^\lambda, \ell), \\ (c, \delta) \leftarrow \mathsf{Poly.Commit}(\mathsf{pp}, \phi) : \\ \mathsf{Poly.VerifyPoly}(\mathsf{pp}, c, \phi, \delta) = T \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Observation: Informally, this say a commitment c to ϕ should verify successfully against ϕ.

Definition 2.5.3 (Evaluation Correctness)**.** $\forall \phi \in \mathbb{F}_p[X]$ with $\deg \phi \leq \ell, \forall i \in \mathbb{F}_p$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Poly.Setup}(1^\lambda, \ell), \\ (c, \delta) \leftarrow \mathsf{Poly.Commit}(\mathsf{pp}, \phi) \\ \pi \leftarrow \mathsf{Poly.ProveEval}(\mathsf{pp}, \phi, i, \delta) : \\ \mathsf{Poly.VerifyEval}(\mathsf{pp}, c, i, \phi(i), \pi) = T \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Observation: Informally, this says an evaluation proof for $\phi(i)$ must verify successfully against a commitment to ϕ.

Definition 2.5.4 (Polynomial Binding)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Poly.Setup}(1^\lambda, \ell), \\ (c, \phi, \phi', \delta, \delta') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ \mathsf{Poly.VerifyPoly}(\mathsf{pp}, c, \phi, \delta) = T \wedge \\ \mathsf{Poly.VerifyPoly}(\mathsf{pp}, c, \phi', \delta') = T \wedge \\ \phi \neq \phi' \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This prevents an adversary from opening the same commitment c to two different polynomials ϕ and ϕ'.

Definition 2.5.5 (Evaluation Binding)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Poly.Setup}(1^\lambda, \ell), \\ (c, i, y, y', \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ \mathsf{Poly.VerifyEval}(\mathsf{pp}, c, i, y, \pi) = T \wedge \\ \mathsf{Poly.VerifyEval}(\mathsf{pp}, c, i, y', \pi') = T \wedge \\ y \neq y' \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This definition prevents an adversary from producing contradicting proofs π and π' that verify against a commitment c, attesting to different evaluations $\phi(i) = y$ and $\phi(i) = y'$ with $y' \neq y$, where $\phi(x)$ is the polynomial committed in c.

Definition 2.5.6 (Computational Hiding)**.** $\forall$ adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Poly.Setup}(1^\lambda, \ell), \phi \in_R \mathbb{F}_p[X] \text{ with } \deg\phi \leq \ell, \\ (c, \delta) \leftarrow \mathsf{Poly.Commit}(\mathsf{pp}, \phi), \\ \left(\mathsf{state}, (x_i)_{i\in[\deg\phi]}\right) \leftarrow \mathcal{A}_0(1^\lambda, \mathsf{pp}, c, \deg\phi) \text{ with distinct } x_i\text{'s}, \\ (\pi_i \leftarrow \mathsf{Poly.ProveEval}(\mathsf{pp}, \phi, x_i, \delta))_{i\in[\deg\phi]}, \\ (\hat{x}, \hat{y}) \leftarrow \mathcal{A}_1\left(1^\lambda, \mathsf{state}, \mathsf{pp}, c, (x_i, \phi(x_i), \pi_i)_{i\in[\deg\phi]}\right) : \\ \hat{y} = \phi(\hat{x}) \wedge \hat{x} \notin \{x_i\}_{i\in[\deg\phi]} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This definition prevents an adversary who gets $\deg\phi$ evaluations $(\phi(x_i))_{i\in[\deg\phi]}$ of a polynomial ϕ from learning another evaluation $\phi(\hat{x})$ at a different point $\hat{x} \neq x_i, \forall i \in [\deg\phi]$ (and thus from learning ϕ itself).

2.5.2 Kate-Zaverucha-Goldberg (KZG) Commitments

Kate, Zaverucha and Goldberg proposed a constant-sized polynomial commitment scheme based on the ℓ-S(B)DH assumption (see Definitions 2.1.2 and 2.1.3). Importantly, evaluation proofs are constant-sized and constant-time to verify; they do not depend in any way on the degree of the committed polynomial. In contrast, both Pedersen-style [173] and Feldman-style [89] commitments are linearly-sized in the degree of the committed polynomial and evaluation proofs take linear time to verify (although the proof size is zero). To provide succinctness, KZG requires ℓ-SDH [29] public parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(g;\tau) = (g^{\tau^i})_{i\in[0,\ell]}$ where τ denotes a trapdoor. (These parameters are computed via a *trusted setup*; see Section 2.6.) Their scheme is computationally-hiding (see Definition 2.5.6) under the discrete log assumption and computationally-binding (see Definition 2.5.4) under ℓ-SDH [128].

Unlike Pedersen commitments [173], KZG can only commit to polynomials of degree $d \leq \ell$. Let ϕ denote a polynomial of degree d with coefficients $c_0, c_1, \ldots, c_d$ in $\mathbb{F}_p$. A KZG commitment to ϕ is a single group element $C = \prod_{i=0}^{d}\left(g^{\tau^i}\right)^{c_i} = g^{\sum_{i=0}^{d} c_i\tau^i} = g^{\phi(\tau)}$. Note that committing to ϕ takes $\Theta(d)$ time. To compute an *evaluation proof* that $\phi(a) = y$, KZG

leverages the polynomial remainder theorem which says

$$\phi(a) = y \Leftrightarrow \exists q(\cdot), \phi(x) - y = q(x)(x - a) \tag{2.3}$$

The proof is just a KZG commitment to q: a single group element $\pi = g^{q(\tau)}$. Computing the proof takes $\Theta(d)$ time. To verify π, one checks (in constant time) if

$$\begin{aligned} e(C/g^y, g) &\stackrel{?}{=} e(\pi, g^\tau/g^a) \Leftrightarrow \\ e(g^{\phi(\tau)-y}, g) &\stackrel{?}{=} e(g^{q(\tau)}, g^{\tau-a}) \Leftrightarrow \\ e(g, g)^{\phi(\tau)-y} &\stackrel{?}{=} e(g, g)^{q(\tau)(\tau-a)} \Leftrightarrow \\ \phi(\tau) - y &\stackrel{?}{=} q(\tau)(\tau - a) \end{aligned}$$

2.5.2.1 Batch proofs and homomorphism

Given a set of points S and their evaluations $\{\phi(i)\}_{i\in S}$, KZG can prove all evaluations with *one* constant-sized *batch proof* rather than $|S|$ individual proofs [128]. The prover computes an *accumulator polynomial* $a(x) = \prod_{i\in S}(x - i)$ in $\Theta(|S| \log^2 |S|)$ time and computes ϕ/a in $\Theta(d \log d)$ time, obtaining a quotient q and remainder r. The batch proof is $\pi = g^{q(\tau)}$.

To verify π against $\{\phi(i)\}_{i\in S}$ and C, the verifier first computes a from S and interpolates r such that $r(i) = \phi(i), \forall i \in S$ in $\Theta(|S| \log^2 |S|)$ time. Next, he computes $g^{a(\tau)}$ and $g^{r(\tau)}$ commitments. Finally, he checks if $e(C/g^{r(\tau)}, g) = e(g^{q(\tau)}, g^{a(\tau)})$. We stress that batch proofs are only useful when $|S| \leq d$. Otherwise, if $|S| > d$, we can interpolate ϕ directly from the evaluations, which makes verifying any evaluation trivial.

Finally, KZG proofs have a *homomorphic* property. Suppose we have two polynomials ϕ_1, ϕ_2 with commitments C_1, C_2 and two proofs π_1, π_2 for $\phi_1(a)$ and $\phi_2(a)$, respectively. Then, a commitment C to the sum polynomial $\phi = \phi_1 + \phi_2$ can be computed as $C = C_1C_2 = g^{\phi_1(\tau)}g^{\phi_2(\tau)} = g^{\phi_1(\tau)+\phi_2(\tau)} = g^{(\phi_1+\phi_2)(\tau)} = g^{\phi(\tau)}$. Even better, a proof π for $\phi(a)$ w.r.t. C can be aggregated as $\pi = \pi_1\pi_2$. This homomorphism is necessary in KZG-based protocols such as eJF-DKG (see Section 8.5).

2.6 Trusted Setup for q-type Assumptions

Our constructions in Parts I and II are secure under the ℓ-S(B)DH (see Definitions 2.1.2 and 2.1.3) and ℓ-PKE assumptions (see Definition 2.1.4) which require *public parameters* of the form:

$$\begin{aligned} \mathsf{PP}_\ell^{\mathsf{SDH}}(g; \tau) &= \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell} \rangle \\ \mathsf{PP}_\ell^{\mathsf{PKE}}(g; \tau, \beta) &= \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell}, g^{\beta\tau}, g^{\beta\tau^2}, \ldots, g^{\beta\tau^\ell} \rangle \end{aligned}$$

This begs the question: Who generates these public parameters? If the answer were "a single party generates them," then that party would have to be trusted to safely discard and forget the trapdoor τ. This would create a single point of failure, which we would like to avoid.

Ideally, a multi-party computation (MPC) protocol [106] should be used to generate the

parameters. The protocol would be run by n parties and guarantee that, as long as one party discards their "share" of τ, the other $n-1$ parties cannot recover τ. Fortunately, previous work [37, 38] shows how to do exactly this for SNARK public parameters [111, 112]. Since our public parameters are a "subset" of SNARK parameters, we can leverage simplified versions of these protocols. In particular, the protocol from [38] makes participation very easy: it allows any number of players to join, participate and optionally drop out of the protocol. In contrast, the first protocol [37] required a fixed, known-in-advance set of players.

Finally, the practicality of such MPC protocols has already been demonstrated. In 2016, six participants used [37] to generate public parameters for the Sprout version of the Zcash cryptocurrency [120]. Two years later, nearly 200 participants used [38] to generate new public parameters for the newer Sapling version of Zcash.

Part I

Append-only Authenticated Data Structures

Chapter 3

Introduction

Scheme	Space	Public parameters	Append time	Lookup proof size	Append-only proof size
Sorted trees [147, 183]	$n \log n$	1	$\log n$	$\log n$	n
Unsorted trees [64, 137]	n	1	$\log n$	n	$\log n$
$\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ (see Section 6.3.2)	λn	$4\lambda n$	$\lambda \log^3 n$	$\log^2 n$	$\log n$
$\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$ (see Section 6.3.2)	λn	1	$\lambda \log^3 n \log\log n$	$\log n$	$\log n$

Table 3.1: Asymptotic costs of our accumulator-based constructions versus previous work. n is the number of key-value pairs in the dictionary and λ is the security parameter. Our AAD constructions asymptotically (and concretely) reduce proof sizes but at the cost of higher space on the server, larger public parameters and slower append times. Our $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ construction additionally requires a trusted setup (see Section 2.6). As a trade-off, our $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$ scheme can reduce public parameters to $2\lambda n$ for $\lambda n \log n$ space.

Security is often bootstrapped from a *public-key infrastructure (PKI)*. For example, on the web, *Certificate Authorities (CAs)* digitally sign *certificates* that bind a website to its public key. This way, a user who successfully verifies the certificate can set up a secure channel with the website. In general, many systems require a PKI or assume one exists [87, 88, 140, 178]. Yet, despite their necessity, PKIs have proven difficult to secure as evidenced by past CA compromises [4, 145, 161].

To address such attacks, *transparency logs* [64, 83, 137] have been proposed as a way of building *accountable* (and thus more secure) PKIs. A transparency log is a *dictionary* managed by an untrusted *log server*. The server periodically appends *key-value pairs* to the dictionary and is queried by mutually-distrusting *users*, who want to know certain keys' values. For example, in *key transparency* [16, 43, 131, 137, 147, 183, 199, 213], CAs are required to publicly log certificates they issue (i.e., values) for each domain (i.e., keys). Fake certificates can thus be detected in the log and CAs can be held accountable for their misbehavior.

Transparency logging is becoming increasingly important in today's Internet. This is evident with the widespread deployment of Google's Certificate Transparency (CT) [137] project. Since its initial March 2013 deployment, CT has publicly logged over 2.1 billion certificates [109]. Furthermore, since April 2018, Google's Chrome browser requires all new

certificates to be published in a CT log [194]. In the same spirit, there has been increased research effort into *software transparency* schemes [7,86,119,162,197,200] for securing software updates. Furthermore, Google is prototyping *general transparency* logs [83, 110] via their Trillian project [110]. Therefore, it is not far-fetched to imagine generalized transparency improving our census system, our elections, and perhaps our government. But to realize their full potential, transparency logs must operate correctly or be easily caught otherwise. Specifically:

Logs Should Prove Inclusion. Transparency logs are meant to expose rogue CAs who issue fake certificates by allowing impersonated victims to easily find such certificates in the log. But this only works if users "enforce" transparency by only accepting certificates that are provably included in the log. To convince users, the log should give them an *inclusion proof* that the certificate is in the log. Otherwise, a rogue CA can simply present the fake certificate directly to the victim user. This makes detecting impersonation attacks much more difficult, since the impersonated victim will not find the fake certificate in the log.

Logs Should Remain Append-only. In a log-based PKI, a devastating attack is still possible: a malicious CA can publish a fake certificate in the log but later collude with the log server to have it removed, which prevents the victim from ever detecting the attack. Transparency logs should therefore prove that they remain *append-only*, i.e., the new version of the log still contains all entries of the old version. One trivial way to provide such a proof is to return the newly-added entries to the user and have the user enforce a subset relation. But this is terribly inefficient. Ideally, a user with a "short" digest h_{old} should accept a new digest h_{new} only if it comes with a succinct *append-only proof* computed by the log. This proof should convince the user that the old log with digest h_{old} is a subset of the new log with digest h_{new}.

Logs Should Support Lookups. When users have access to digests (instead of whole logs), the central question becomes: How can a user check *against their digest* which values are registered for a certain key k in the log? Ideally, a small *lookup proof* should convince the user that the server has returned *nothing more or less than all values* of key k. More formally, the server should *not* be able to equivocate and present one set of values V for k to a user and a different set V' to some other user. In other words, users who have the same digest should see the same set of values for any key k.

Logs Should Remain Fork-Consistent. An unavoidable issue is that a malicious log server can also equivocate about digests and *fork* users [64, 140]. For example, at time i, the server can append (k, v) to one user's log while appending (k, v') to another user's log. Since the two users' logs will differ at location i, their digests will also differ. Intuitively, *fork consistency* [140,141] guarantees that if two users are given two different digests as above, they must forever be given different digests. Fork consistency enables users to *gossip* [62,67,197,200] and detect forks by checking if they are seeing different digests.

Challenges. Building transparency logs with succinct lookup and append-only proofs is a long-standing open problem. At first glance, a Merkle-based [150] solution seems possible. Unfortunately, it appears very difficult to organize a Merkle tree so as to support both succinct append-only proofs and succinct lookup proofs. On one hand, trees with chronologically-ordered leaves [64, 146, 203] support logarithmic-sized append-only proofs but at the cost of linear-sized lookup proofs. On the other hand, trees can be lexicographically-ordered by key [12, 43, 65, 163] to support succinct lookup proofs at the cost of linear append-only proofs (see Section 6.4.2).

It might seem natural to combine the two and obtain succinct lookup proofs via the lexicographic tree and succinct append-only proofs via the chronologic tree [183]. But this does not work either, since there must be a succinct proof that the two trees "correspond": they are correctly built over the same set of key-value pairs. While previous transparency logs [183, 213] work around this by having users "collectively" verify that the two trees correspond [56, 183, 213], this requires a sufficiently high number of honest users and can result in slow detection. An alternative, which we discuss in Section 11.1.2.3, is to use SNARKs [95, 112] but this is expensive.

At second glance, *cryptographic accumulators* [22,160] seem useful for building transparency logs (see Section 2.5). Unfortunately, accumulators are asymptotically-inefficient, requiring quadratic time to compute all proofs or to update all proofs after a change to the set. As a result, a computationally-efficient accumulator-based solution is not obvious.

Our Contribution. We introduce a novel cryptographic primitive called an *append-only authenticated dictionary (AAD)* that captures the functionality of a secure transparency log. Put simply, an AAD maps a key to one or more values in an append-only fashion. We are the first to give security definitions for AADs. We are also the first to instantiate asymptotically *efficient* AADs from *bilinear accumulators* [160] and *RSA accumulators* (see Section 6.3). Importantly, our design does not rely on collective verification by users nor on trusted third parties, assuming only an untrusted log server. Our AAD offers logarithmic-sized append-only proofs, logarithmic-sized lookup proofs and polylogarithmic worst-case time appends (see Table 3.1).

We implement one of our AAD constructions based on bilinear accumulators in C++ and evaluate it. Our lookup and append-only proofs are in the order of tens of KiBs and our verification time is in the order of seconds. For example, a proof for a key with 32 values in a dictionary of 10^6 entries is 94 KiB and verifies in 2.5 seconds. While our lookup proof sizes are larger than in previous work, our small-sized append-only proofs can help significantly reduce the overall communication in transparency logs, as we show in Section 6.4.2.1. Our code is available at:

`https://github.com/alinush/libaad-ccs2019`

3.1 Overview of Techniques

We first build an efficient *append-only authenticated set* (AAS), instead of an AAD. An AAS is an append-only set of elements with proofs of (non)membership of any element.

In other words, an AAS is just a universal accumulator [139] with subset witnesses [170] that additionally offers fork consistency. The challenge, however, is to support fast appends while precomputing all proofs. We overcome this challenge and give AAS constructions from bilinear accumulators (see Section 4.3.1) and RSA accumulators (see Section 4.3.2).

However, to efficiently implement *any* transparency log, we must modify our AAS into an AAD, which is more "expressive." Specifically, an AAD can provably return all values of a key, while an AAS can only prove that an element is or is not in the set. In Chapter 6, we describe two techniques to change our AAS into an AAD. Our two techniques trade off space on the server for faster append times and less public parameters.

AAS From Bilinear Accumulators. As explained above, a bilinear accumulator almost fully implements an AAS. However, bilinear accumulators are computationally-expensive. Specifically, updating the set and computing a single (non)membership witness takes time linear in n, the size of the set. Thus, computing all n membership witnesses in a bilinear accumulator would take $\Theta(n^2)$, which is prohibitive. Our work reduces these times to polylogarithmic, but at the cost of increasing witness size from constant to $O(\log^2 n)$.

First, we introduce *Communion Trees* (CTs), a hierarchical way to precompute all membership witnesses in a bilinear accumulator in $\Theta(n \log^2 n)$ time (instead of $O(n^2)$). Second, instead of "accumulating" the elements directly, we build a "sparse" prefix tree (or trie) over all elements and accumulate the tree itself. Then, we precompute non-membership witnesses for all prefixes at the *frontier* of this tree (see Figure 5-3) in quasilinear time. As a result, non-membership of an element is reduced to non-membership of one of its prefixes. (This frontier technique was originally proposed in [152].) Finally, we use classic amortization techniques [165, 166] to append in polylogarithmic time and to precompute append-only proofs between any version i and j of the set.

AAS From RSA Accumulators. By replacing the bilinear accumulator with an RSA accumulator in our bilinear-based AAS, we can obtain an RSA-based AAS. However, for this to work, we must first enhance RSA accumulators with *subset witnesses* and *disjointness witnesses* (see Section 5.4.1). The RSA-based AAS has a few advantages over the bilinear-based one. First, all constant-sized membership witnesses in an RSA accumulator can be computed in $\Theta(n \log n)$ time (see Section 4.3.2). This asymptotically reduces our AAS (and AAD) proof sizes. Second, RSA accumulators have constant-sized public parameters. This reduces storage and memory on the server.

Limitations. We only implemented one of our bilinear-based AADs called $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ (see Section 6.3.2), since group operations are faster in bilinear groups than in RSA groups. A key limitation is that $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ has high append times (i.e., a few seconds per append) and high memory usage (i.e., hundreds of GiBs for an AAD of size 2^{20}). However, we discuss how to improve it in Sections 6.4.1.1, 6.4.1.4 and 11.1.

Another limitation is that $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ security relies on the q-PKE "knowledge" assumption (commonly used in SNARKs [101, 111]). Hence, we need a large set of public parameters that must be generated via a *trusted setup* phase, which complicates deployment. We discussed how this trusted setup can be decentralized in Section 2.6. Our RSA-based construction

is proven secure in the generic group model in hidden-order groups [69] but only requires constant-sized parameters, which must also be generated via a trusted setup (see Section 4.4).

3.2 Related Work

The key difference between AADs and previous work [16, 43, 131, 137, 147, 183, 199, 213] is that we offer succinct proofs for everything while only relying on a single, untrusted log server. In contrast, previous work either has large proofs [137, 147], requires users to "collectively" verify the log [183, 213] (which assumes enough honest users and can make detection slow), or makes some kind of trust assumption about one or more actors [16, 131, 137, 199]. On the other hand, previous work only relies on collision-resistant hash functions (CRHFs), which makes it computationally much cheaper. However, since communication is more expensive than computation, this is not always the right trade-off.

In contrast, our bilinear constructions require trusted setup, large public parameters, and non-standard assumptions. Our RSA-based constructions eliminate some of these drawbacks, requiring no trusted setup and having constant-sized public parameters. Unlike previous work, our constructions are not yet practical due to their high append times and memory usage (see Sections 6.4.1.1 and 6.4.1.4). Finally, previous work [16, 131, 199, 213] addresses in more depth the subtleties of log-based PKIs, while our work is focused on improving the transparency log primitive itself by providing succinct proofs with no trust assumptions.

(E)CT. Early work proposes the use of Merkle trees for public-key distribution but does not tackle the append-only problem, only offering succinct lookup proofs [43, 132, 158]. Accumulators are dismissed in [43] due to trusted setup requirements. Certificate Transparency (CT) [137] provides succinct append-only proofs via *history trees* (HTs). Unfortunately, CT does not offer succinct lookup proofs, relying on users to download each update to the log to discover fake PKs, which can be bandwidth-intensive (see Section 6.4.2.1). Alternatively, users can look up their PKs via one or more CT *monitors*, who download and index the entire log. But this introduces a trust assumption that a user can reach at least one honest CT monitor. Enhanced Certificate Transparency (ECT) addresses CT's shortcomings by combining a lexicographic tree with a chronologic tree, with collective verification by users. As discussed above, collective verification assumes enough honest users exist and can make detection of append-only breaks slow. Alternatively, ECT can also rely on one or more "public auditors" to verify correspondence of the two trees, but this introduces a trust assumption.

A(RP)KI and PoliCert. Accountable Key Infrastructure (AKI) [131] introduces a checks-and-balances approach where log servers manage a lexicographic tree of certificates and so-called "validators" ensure log servers update their trees in an append-only fashion. Unfortunately, AKI must *"assume a set of entities that do not collude: CAs, public log servers, and validators"* [131]. At the same time, an advantage of AKI is that validators serve as nodes in a gossip protocol, which helps detect forks. ARPKI [16] and PoliCert [199] extend AKI by providing security against attackers controlling $n - 1$ out of n actors. Unfortunately, this means ARPKI and PoliCert rely on an anytrust assumption to keep their logs append-only. On the other hand, AKI, ARPKI and PoliCert carefully consider many of the intricacies

of PKIs in their design (e.g., certificate policies, browser policies, deployment incentives, interoperability). In addition, ARPKI formally verifies their design.

CONIKS and DTKI. CONIKS [147] periodically appends a digest of a prefix tree to a hash chain. However, users must collectively verify the tree remains append-only. Specifically, *in every published digest*, each user checks that their own public key has not been removed or maliciously changed. Unfortunately, this process can be bandwidth-intensive when digests are published frequently (see Section 6.4.2.1). DTKI [213] observes that relying on a multiplicity of logs (as in CT) creates overhead for domain owners who must check for impersonation in every log. DTKI introduces a *mapping log* that associates sets of domains to their own exclusive transparency log. Unfortunately, like ECT, DTKI also relies on users to collectively verify its many logs. To summarize, while previous work [16, 131, 199, 213] addresses many facets of the transparent PKI problem, it does not address the problem of building a transparency log with succinct proofs without trust assumptions and without collective verification.

Byzantine Fault Tolerance (BFT). If one is willing to move away from the single untrusted server model, then a transparency log could be implemented using BFT protocols [53, 135, 155]. In fact, BFT can trivially keep logs append-only and provide lookup proofs via sorted Merkle trees. With permissioned BFT [53], one must trust that two thirds of BFT servers are honest. While we are not aware of permissioned implementations, they are worth exploring. For example, in the key transparency setting, it is conceivable that CAs might act as BFT servers. With permissionless BFT [155, 211], one needs a cryptocurrency secured by proof-of-work or proof-of-stake. Examples of this are Namecoin [156], Blockstack [9] and EthIKS [35].

Formalizations. Previous work formalizes Certificate Transparency (CT) [57, 79] and general transparency logs [57]. In contrast, our work formalizes append-only authenticated dictionaries (AAD) and sets (AAS), which can be used as transparency logs. Our AAD abstraction is more expressive than the *dynamic list commitment (DLC)* abstraction introduced in [57]. Specifically, DLCs are append-only lists with non-membership by insertion time, while AADs are append-only dictionaries with non-membership by arbitrary keys. Nonetheless, AADs can easily associate each key-value pair to its insertion time securely via Merkle aggregation [64]. This way, AADs can easily support non-membership by insertion time as well. Finally, previous work carefully formalizes proofs of misbehavior for transparency logs [57, 79]. Although misbehavior in AADs is provable too, we do not formalize this in the paper.

Neither our work nor previous work adequately models the network connectivity assumptions needed to detect forks in a gossip protocol. Lastly, previous work improves or extends transparency logging in various ways but does not tackle the append-only problem [66,85,174].

Chapter 4

Preliminaries

Notation. We introduce some helpful notation for (authenticated) sets and dictionaries, which are the focus of this part of the thesis. Let $\mathsf{Primes}(\lambda)$ denote the set of all λ-bit prime numbers. Let $|S|$ denote the number of elements in a multiset S (e.g., $S = \{1, 2, 2\}$ and $|S| = 3$). For dictionaries, let $\mathcal{K}$ be the set of all possible keys and $\mathcal{V}$ be the set of all possible values. ($\mathcal{K}$ and $\mathcal{V}$ are application-specific; e.g., in software transparency, a key is the software package name and a value is the hash of a specific version of this software package.) Let $\mathcal{P}(\mathcal{V})$ denote all possible multisets with elements from $\mathcal{V}$. Let $K \subset \mathcal{K}$. Then, a *dictionary* is a function $D : K \to \mathcal{P}(\mathcal{V})$ that maps a key $k \in K$ to a multiset of values $V \in \mathcal{P}(\mathcal{V})$ (including the empty set). Let $D(k)$ denote the multiset of values associated with key k in dictionary D. Let $|D|$ denote the number of key-value pairs in the dictionary. We also refer to this as the *version* of the dictionary. Appending (k, v) to a version i dictionary updates the multiset $V = D(k)$ of key k to $V' = V \cup \{v\}$ and increments the dictionary version to $i + 1$. Finally, let ε denote the empty string.

4.1 Cryptographic Assumptions

Some of the results in this thesis (see Section 5.4) are based on cryptosystems which use *hidden-order groups* such as $\mathbb{Z}_N^* = \{a \mid \gcd(a, N) = 1\}$ where $N = pq$ is the product of two primes [182]. Let $\mathbb{G}_? \leftarrow \mathcal{G}_{\mathsf{hidden}}(1^\lambda)$ denote an algorithm for generating such groups, where λ is a security parameter. In addition to classical assumptions, such as the *discrete log (DL) assumption* and the *RSA assumption*, this thesis relies on the *Strong RSA assumption* and the *Adaptive Root assumption*, which we describe below.

4.1.1 Strong RSA Assumption

Definition 4.1.1 (Strong RSA Assumption)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathbb{G}_? \leftarrow \mathcal{G}_{\mathsf{hidden}}(1^\lambda), g \in_R \mathbb{G}_?, \\ (u, e) \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}_?, g) : \\ u^e = g \text{ and } e \text{ is prime} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Informally, this assumption says that no computationally-bounded adversary can compute

any eth prime root of a random element g. This is a generalization of the RSA assumption which says that, for a *fixed* e, no computationally-bounded adversary can compute an eth root of a random g.

4.1.2 Adaptive Root Assumption

This assumption was introduced by Wesolowski [209].

Definition 4.1.2 (Adaptive Root Assumption)**.** $\forall$ adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathbb{G}_? \leftarrow \mathcal{G}_{\mathsf{hidden}}(1^\lambda), \\ (w, \mathsf{state}) \leftarrow \mathcal{A}_0(1^\lambda, \mathbb{G}_?), \\ e \in_R \mathsf{Primes}(\lambda), \\ u \leftarrow \mathcal{A}_1(1^\lambda, e, \mathsf{state}) : \\ u^e = w \wedge w \neq 1 \end{array}\right] \leq \mathsf{negl}(\lambda)$$

For this assumption to hold, there cannot be any known-order elements in $\mathbb{G}_?$. This means $\mathbb{G}_? = \mathbb{Z}_N^*$ should **not** be used directly. Instead, $\mathbb{Z}_N^* \backslash \{\pm 1\}$ should be used, as described in [31]. Finally, note that the Adaptive Root Assumption and the Strong RSA Assumption (see Definition 4.1.1) are incomparable. In the former, the adversary picks the base and is asked to compute a specific, randomly-picked root of that base. In the latter, the adversary is randomly given a base and asked to compute *any* prime root of that base.

4.2 Proofs (of Knowledge) of Exponentiation in Hidden-Order Groups

In this section, we introduce *proofs of knowledge of exponent*, which we use in Section 5.4 to build append-only authenticated sets and dictionaries. Informally, these protocols allow a prover to convince a verifier that the prover knows a potentially-large x such that $g^x = y$ without having to reveal x. All protocols in this section are proven secure in [32] under the adaptive root assumption [209] and in the generic group model for hidden-order groups [69], except for PoE (see Section 4.2.1) which does not use the generic group model. All of the protocols explained in this section are *interactive* but can be easily made non-interactive via the Fiat-Shamir heuristic in the random oracle model (ROM) [90].

4.2.1 Proof of Exponentiation (PoE)

Wesolowski introduced a *proof of exponentiation* (PoE) protocol for checking that an exponentiation $u^x = w$ with $x = 2^t$ was performed correctly in a hidden-order group. Importantly, this protocol allows a verifier to check much faster than naively redoing the exponentiation [209]. When x is very large, as often happens in applications such as RSA accumulators, this protocol saves verifiers a significant amount of time. Boneh et al. [31] show Wesolowski's protocol generalizes to any $x \in \mathbb{Z}$. We detail their protocol below. (Recall from Chapter 4

that $\mathsf{Primes}(\lambda)$ denotes the set of all λ-bit prime numbers.)

Input: $u, w \in \mathbb{G}_?$, $x \in \mathbb{Z}$
Claim: $u^x = w$

- Verifier sends $\ell \in_R \mathsf{Primes}(\lambda)$ to prover.
- Prover divides x by ℓ, obtaining integers $q \in \mathbb{Z}, r \in [0, \ell)$ such that $x = q\ell + r$.
- Prover sends proof $Q = u^q \in \mathbb{G}_?$ to the verifier.
- Verifier computes $r = (x \bmod \ell) \in [0, \ell)$ and accepts proof if $Q^\ell u^r = w$.

This PoE protocol is proven secure in [209] under the Adaptive Root Assumption in hidden-order groups (see Definition 4.1.2).

Computational Cost. The prover needs to divide an $|x|$-bit number by a λ-bit number and exponentiate by a $(|x| - \lambda)$-bit quotient q. The verifier must (1) pick a random λ-bit prime, which involves multiple expensive primality tests (2) compute a big division of an $|x|$-bit number by a λ-bit number and (3) do two exponentiations with λ-bit exponents. When made non-interactive via the Fiat-Shamir heuristic [90], the cost of picking a random λ-bit prime is shifted to the prover.

4.2.2 Proof of Knowledge of Exponent for Fixed Base g (PoKE*)

Boneh et al. [31] transform the PoE protocol from above into a *proof of knowledge* [108] of an x such that $g^x = w$. They call this protocol PoKE*. A caveat is that PoKE* requires a *common reference string* (CRS) that fixes the base g. This restriction is later removed in Section 4.2.4.

Common Reference String (CRS): A generator g of $\mathbb{G}_?$
Input: $w \in \mathbb{G}_?$, $x \in \mathbb{Z}$
Claim: $g^x = w$

- Verifier sends $\ell \in_R \mathsf{Primes}(\lambda)$ to prover.
- Prover divides x by ℓ, obtaining integers $q \in \mathbb{Z}, r \in [0, \ell)$ such that $x = q\ell + r$.
- Prover sends proof $(Q = g^q, r)$ to verifier.
- Verifier accepts proof if (1) $r \in [0, \ell)$ and (2) $Q^\ell g^r = w$.

The PoKE* protocol is proven secure in the generic group model for hidden-order groups (see "Proof of Theorem 7" in Appendix C.2 in [32]).

Computational Cost. For the prover, the cost remains the same as in PoE (see Section 4.2.1). However, the cost decreases for the verifier, who no longer has to divide x by ℓ.

4.2.3 Proof of Knowledge of Exponent for Any Base u (PoKE)

As pointed out in [31], the PoKE* protocol from Section 4.2.3 only works for group generators g that are part of the *common reference string* (CRS). In other words, it does not work for proving knowledge of an x such that $u^x = w$ for *arbitrary* $u \in \mathbb{G}_?$. To see why, consider an attacker who sets $u = g^a$ and $w = g^b$ with $b \neq 1$ but who cannot compute the discrete log $x = \log_u(w) = a/b$ because he cannot invert b. Clearly, such an attacker should not be able to construct a valid PoKE* proof for knowing $x = a/b$, yet he can do so as follows.

First note that, with overwhelming probability, the challenge ℓ is co-prime with a. Thus, the attacker can compute Bézout coefficients q', r' such that $q'\ell + r'a = 1$. Then, he can compute $q = bq', r = br'$ such that $q\ell + ra = b$. This implies that:

$$q\ell + ra = b \Leftrightarrow \tag{4.1}$$
$$g^{q\ell+ra} = g^b \Leftrightarrow \tag{4.2}$$
$$(g^q)^\ell (g^a)^r = w \Leftrightarrow \tag{4.3}$$
$$(g^q)^\ell u^r = w \tag{4.4}$$

Notice that Equation (4.4) exactly matches the PoKE* proof verification equation in Section 4.2.2. Thus, the attacker has successfully forged a PoKE* proof $(Q = g^q, r)$ for knowing the discrete log $x = \log_u(w)$.

To fix this, Boneh et al. propose a new PoKE protocol [31]. The key idea is to have the prover do two PoKE* proofs in parallel: one for $u^x = w$ and one for $g^x = z$. This protocol is detailed below.

Common Reference String (CRS): A generator g of $\mathbb{G}_?$
Input: $u, w \in \mathbb{G}_?$, $x \in \mathbb{Z}$
Claim: $u^x = w$

- Prover sends $z = g^x$ to the verifier.
- Verifier sends $\ell \in_R \mathsf{Primes}(\lambda)$ to prover.
- Prover divides x by ℓ, obtaining integers $q \in \mathbb{Z}, r \in [0, \ell)$ such that $x = q\ell + r$.
- Prover sends proof $(Q = u^q, Q' = g^q, r)$.
- Verifier accepts proof if (1) $r \in [0, \ell)$, (2) $Q^\ell u^r \stackrel{?}{=} w$ and (3) $Q'^\ell g^r \stackrel{?}{=} z$.

PoKE proofs are 3 group elements (z, Q, Q') and one ℓ-bit integer r, while PoKE* proofs are just one group element and one ℓ-bit integer. The next PoKE2 protocol reduces the proof size and also removes the need for a CRS. PoKE is proven secure in the generic group model where the adaptive root assumption holds (see "Proof of Theorem 3" in Appendix C.2 in [32]).

Computational Cost. Compared to PoKE*, the prover cost increases by two big $O(|x|)$-bit exponentiations (for computing z and Q') and the verifier cost increases by two small λ-bit exponentiations (for computing Q'^ℓ and g^r).

4.2.4 PoK of Exponent for Any Base u Without a CRS (PoKE2)

This protocol reduces PoKE's proof size and eliminates the CRS by having the verifier pick g randomly. It also results in a 2× faster prover, at the cost of making verification 1.5× slower. This is a good trade-off for applications where proving time, not verification time, is the bottleneck (e.g., our AAS and AAD in Sections 5.4 and 6.3). The intuition behind PoKE2 is to combine the two PoKE* proofs for g^x and u^x into a single one via a linear combination with a random α.

Input: $u, w \in \mathbb{G}_?$, $x \in \mathbb{Z}$
Claim: $u^x = w$

- Verifier sends $g \in_R \mathbb{G}_?$ to prover.
- Prover sends $z = g^x$ to the verifier.
- Verifier sends $\ell \in_R \mathsf{Primes}(\lambda)$ and $\alpha \in [0, 2^\lambda)$ to prover.
- Prover divides x by ℓ, obtaining integers $q \in \mathbb{Z}, r \in [0, \ell)$ such that $x = q\ell + r$.
- Prover sends proof $(Q = (ug^\alpha)^q, r)$.
- Verifier accepts proof if (1) $r \in [0, \ell)$ and (2) $Q^\ell u^r g^{\alpha r} \stackrel{?}{=} wz^\alpha$

Note that:

$$\begin{aligned} Q^\ell u^r g^{\alpha r} &= (u^q g^{\alpha q})^\ell u^r g^{\alpha r} \\ &= u^{q\ell} g^{\alpha q \ell} u^r g^{\alpha r} \\ &= u^{q\ell + r} g^{\alpha q \ell + \alpha r} \\ &= u^x g^{\alpha x} \\ &= w z^\alpha \end{aligned}$$

The proof is (z, Q, r): two group elements and a λ-bit number r. Finally, the PoKE2 protocol is proven secure in the generic group model (see "Proof of Theorem 3" in Appendix C.2 in [32]).

Computational Cost. Compared to PoKE* (see Section 4.2.2), the prover cost increases by one big $O(|x|)$-bit exponentiations (for computing z) and one small λ-bit exponentiation (for computing g^α). This is 2× more efficient than the PoKE prover from Section 4.2.3. The verifier cost increases by three small λ-bit exponentiations (for $(g^\alpha)^r$ and for z^α). This is 1.5× slower than the PoKE verifier.

4.3 Cryptographic Accumulators

A *cryptographic accumulator* [22,160] is a commitment to a set of *elements* $T = \{e_1, e_2, \ldots, e_n\}$ referred to as the *accumulated set*. This commitment is often referred to as the *accumulator*. Accumulators have three key features. First, *membership* and *non-membership* of any element can be proven w.r.t. the accumulator. These proofs are called *(non)membership witnesses*

and are typically constant-sized. Accumulators that support both membership and non-membership witnesses are called *universal* [139]. Second, given two sets $T_1 \subseteq T_2$ and their accumulators a_1 and a_2, a *subset witness* can be computed that convinces anyone with a_1 and a_2 that $T_1 \subseteq T_2$. Third, a *disjointness witness* can be computed for sets $T_1 \cap T_2 = \varnothing$ w.r.t. their accumulators.

A *Merkle hash tree* over n elements can be considered to be a cryptographic accumulator since, depending on how it is organized, it can offer some of the accumulator functionality (but not all). In this thesis, however, we are concerned with accumulators that offer *all* of the features from above. This limits us to *RSA accumulators* (see Section 4.3.2), which we enhance in Section 5.4.1, and *bilinear accumulators* (see Section 4.3.1).

4.3.1 Bilinear Accumulators

Bilinear accumulators were introduced by Nguyen [160]. Damgård and Triandopoulos later extended them with non-membership witnesses [71]. Canard and Gouget introduce subset witnesses, but only prove the security of the ecash scheme built on top of the witnesses [49]. Papamanthou et al. [170] later formalized and instantiated subset, intersection and set difference witnesses. They proved security of these witnesses under ℓ-SBDH (see Definition 2.1.3).

Setup. Bilinear accumulators are *bounded*, which means they can only commit to sets of at most ℓ elements. Their setup involves two steps. First, a prime-order group with bilinear maps must be generated (see Section 2.1.1). Second, one must generate ℓ-SDH public parameters:

$$\mathsf{PP}_{\ell}^{\mathsf{SDH}}(g;\tau) = \langle g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^{\ell}} \rangle \text{ where } \tau \in_R \mathbb{Z}_p^*$$

For this, a *trusted setup ceremony* can be used that computes the g^{τ^i} powers and "forgets" τ [37, 38]. Importantly, this ceremony can be implemented as an MPC protocol over multiple parties (see Section 2.6).

Committing to a Set. Let $T = \{e_1, e_2, \ldots, e_n\}$ denote a set of elements. Let $\mathcal{C}_T(x) = (x-e_1)(x-e_2)\cdots(x-e_n)$ denote the *characteristic polynomial* of T. The accumulator $\mathsf{acc}(T)$ of T is computed as $\mathsf{acc}(T) = g^{\mathcal{C}_T(\tau)} = g^{(\tau-e_1)(\tau-e_2)\cdots(\tau-e_n)}$. Given coefficients $c_0, c_1, \ldots, c_n$ of $\mathcal{C}_T(\cdot)$ where $n \leq \ell$, the accumulator is computed *without knowledge* of τ as follows:

$$\mathsf{acc}(T) = g^{c_0}(g^{\tau})^{c_1}(g^{\tau^2})^{c_2}\cdots(g^{\tau^n})^{c_n} = g^{c_0 + c_1\tau + c_2\tau^2 \cdots c_n\tau^n} = g^{\mathcal{C}_T(\tau)}$$

In other words, the server computes a KZG commitment (see Section 2.5.2) to the characteristic polynomial of T. Since the accumulator only supports elements from $\mathbb{F}_p$, we assume an injective function $\mathcal{H}_{\mathbb{F}} : \mathcal{D} \to \mathbb{F}_p$ that maps elements to be accumulated from any domain $\mathcal{D}$ to values in $\mathbb{F}_p$. If $|\mathcal{D}| > |\mathbb{F}_p|$, then $\mathcal{H}_{\mathbb{F}}$ can be a collision-resistant hash function (CRHF).

Membership Witnesses. A *prover* who has T can convince a *verifier* who has $\mathsf{acc}(T)$ that an *element* e_i is in the set T. The prover simply convinces the verifier that $(x - e_i) \mid \mathcal{C}_T(x)$ by presenting a KZG commitment $\pi = g^{q(\tau)}$ to a quotient polynomial $q(\cdot)$ such that

$\mathcal{C}_T(x) = (x - e_i)q(x)$. Using a bilinear map, the verifier checks the property above holds at $x = \tau$.

$$\begin{aligned} e(g, \mathsf{acc}(T)) &\stackrel{?}{=} e(\pi, g^{\tau}/g^{e_i}) \Leftrightarrow \\ e(g,g)^{\mathcal{C}_T(\tau)} &\stackrel{?}{=} e(g,g)^{(\tau - e_i)q(\tau)} \Leftrightarrow \\ \mathcal{C}_T(\tau) &\stackrel{?}{=} (\tau - e_i)q(\tau) \end{aligned}$$

This membership witness can be proven secure under the ℓ-SDH assumption [128]. Bilinear accumulators also support non-membership witnesses [71], but this thesis does not make (direct) use of them.

Subset and Disjointness Witnesses. To prove that $A \subseteq B$, the prover shows that $\mathcal{C}_A(x) \mid \mathcal{C}_B(x)$. Specifically, the prover presents a KZG commitment $\pi = g^{q(\tau)}$ of a quotient polynomial $q(\cdot)$ such that $\mathcal{C}_B(x) = q(x)\mathcal{C}_A(x)$. The verifier checks that $e(g, \mathsf{acc}(B)) = e(\pi, \mathsf{acc}(A))$.

To prove that $A \cap B = \varnothing$, the prover uses the Extended Euclidean Algorithm (EEA) [205] to compute Bézout coefficients $y(\cdot)$ and $z(\cdot)$ such that $y(x)\mathcal{C}_A(x) + z(x)\mathcal{C}_B(x) = 1$. The witness consists of a KZG commitment to the Bézout coefficients $\gamma = g^{y(\tau)}$ and $\zeta = g^{z(\tau)}$. The verifier checks that $e(\gamma, \mathsf{acc}(A))e(\zeta, \mathsf{acc}(B)) = e(g,g)$. By setting $B = \{e\}$, we get a new type of non-membership witness for $e \notin A$, different than the one by Damgård and Triandopoulos [71]. Both subset and disjointness witnesses can be proven secure under q-SBDH (see Definition 2.1.3).

4.3.2 RSA Accumulators

Benaloh and de Mare [22] were the first to introduce the notion of cryptographic accumulators and to instantiate it as *RSA accumulators* via the Strong RSA assumption [14]. Barić and Pfitzmann [14] showed RSA accumulators are not just one-way functions but also collision-resistant. Sander et al. [184] showed how to efficiently precompute all witnesses in RSA accumulators in quasilinear time. Li et al. [139] extended RSA accumulators with non-membership witnesses. Boneh et al. [31] added support for computing and verifying witnesses in batch, for aggregating multiple witnesses into one and for union witnesses.

RSA accumulators have several advantages over bilinear accumulators. First, RSA accumulators have constant-sized parameters, whereas bilinear accumulators need q-SDH public parameters (see Definition 2.1.2) for committing to sets of size $\leq q$. Second, RSA accumulators do not require trusted setup if instantiated over *class groups* [31,144] rather than the integers modulo N where $N = pq$ is the product of two, unknown primes. Third, RSA accumulators support precomputing all constant-sized membership witnesses in quasilinear time, which we discuss below.

RSA accumulators also have a few disadvantages. First, committing to a set of size n involves performing a big, λn-bit sized exponentiation, which is inherently difficult to speed up via multi-threading. (Indeed, this difficulty is leveraged to implement verifiable delay functions (VDFs) [30, 177, 209].) Second, membership witnesses are much larger than in bilinear accumulators (e.g., 256 bytes versus 32 bytes when $|N| = 2048$ bits). Third,

they do not support subset and disjointness witnesses, which we introduce in this thesis in Section 5.4.1.

Setup. Unlike bilinear accumulators (see Section 4.3.1) which work over groups of prime (known) order, RSA accumulators work over *hidden-order groups* such as the subgroup of quadratic residues $\mathbb{QR}_N$ of $\mathbb{Z}_N^*$ where $N = pq$ is the product of two primes. We assume such a hidden-order group $\mathbb{G}_?$ with generator g has been set up. For example, if this group is $\mathbb{QR}_N$, we assume nobody knows the factorization of N. (In Section 4.4, we discuss how this can be achieved.)

Committing to a Set. Let $T = \{e_1, e_2, \ldots, e_n\}$ denote a set of elements. For RSA accumulators, the elements e_i must be *prime numbers*. In particular, we will restrict ourselves to elements that are 2λ-bit prime numbers. A collision-resistant hash function (CRHF) $\mathcal{H}_{\mathsf{prime}}$ can be used to map any elements from any domain to such 2λ-bit primes. To commit to T, one simply computes:

$$\mathsf{acc}(T) = g^{\prod_{e_i \in T} e_i}$$

Membership Witnesses. A membership witness for e_i is just an RSA accumulator over $T \setminus \{e_i\}$:

$$\pi = \mathsf{acc}(T \setminus \{e_i\}) = g^{\prod_{e_j \in T \setminus \{e_i\}} e_j} = \mathsf{acc}(T)^{1/e_i}$$

Importantly, note that the witness π cannot be computed efficiently by exponentiating $\mathsf{acc}(T)$ by $1/e_i$. This is because e_i^{-1} cannot be efficiently computed "in the exponent" of $\mathbb{G}_?$, which is a hidden-order group. Thus, the membership witness must be computed in a less efficient manner as:

$$g^{\prod_{e_j \in T \setminus \{e_i\}} e_j}$$

For this, the prover must do $n-1$ small, 2λ-bit-sized exponentiations, which takes $\Theta(n)$ time. (Alternatively, the prover can do one big, $\lambda(n-1)$-bit-sized exponentiation, which takes the same amount of time.)

To verify the membership witness for e_i, a verifier with $\mathsf{acc}(T)$ checks that:

$$\pi^{e_i} = \mathsf{acc}(T) \Leftrightarrow \tag{4.5}$$

$$\mathsf{acc}(T \setminus \{e_i\})^{e_i} = \mathsf{acc}(T) \Leftrightarrow \tag{4.6}$$

$$\left(\mathsf{acc}(T)^{1/e_i}\right)^{e_i} = \mathsf{acc}(T) \Leftrightarrow \tag{4.7}$$

$$\mathsf{acc}(T) = \mathsf{acc}(T) \tag{4.8}$$

This way, membership witnesses can be verified in constant-time with a single exponentiation. Membership witnesses can be proven secure under the Strong RSA assumption (see Definition 4.1.1). RSA accumulators also support non-membership witnesses [139], but this thesis does not make (direct) use of them.

Membership Witness Aggregation. Boneh et al. [31] show how to *aggregate* two membership witnesses for two distinct elements into a single witness. Specifically, given π_1, π_2

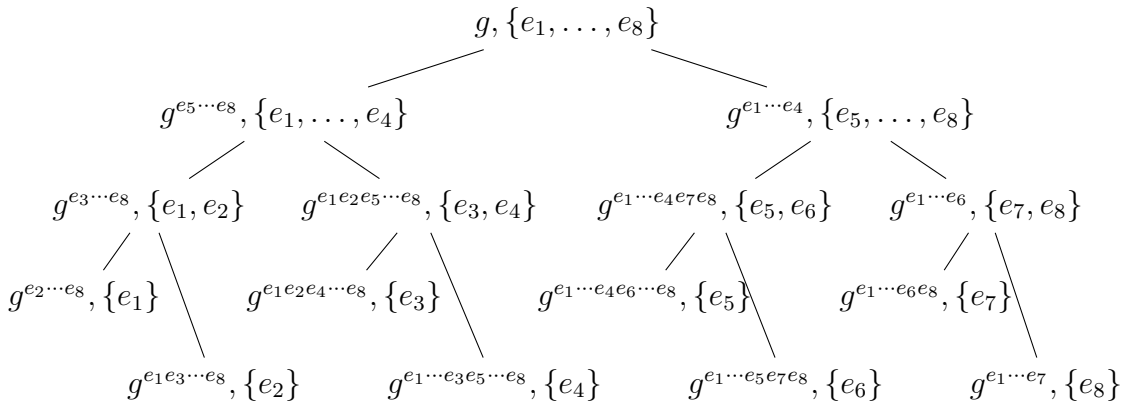


Figure 4-1: Divide-and-conquer approach by Sander et al. [184] for precomputing n membership witnesses for $T = \{e_1, \ldots, e_n\}$ w.r.t. the RSA accumulator $\mathsf{acc}(T)$ in $\Theta(n \log n)$ time. Here $n = 8$.

for elements e_1 and e_2 respectively, a single witness π for the two elements can be obtained as follows. First, Bézout coefficients a, b such that $ae_1 + be_2 = 1$ are calculated using the Extended Euclidean Algorithm (EEA). Then, the aggregated witness is computed as $\pi = \pi_1^b \pi_2^a$. To verify this aggregated witness, the verifier checks that $\pi^{e_1 e_2} = \mathsf{acc}(T)$.

To see why this works, note that π is indeed equal to $\mathsf{acc}(T)^{\frac{1}{e_1 e_2}}$:

$$\begin{aligned}
\pi &= \pi_1^b \pi_2^a \Leftrightarrow \\
&= \left(\mathsf{acc}(T)^{1/e_1}\right)^b \left(\mathsf{acc}(T)^{1/e_2}\right)^a \Leftrightarrow \\
&= \mathsf{acc}(T)^{b/e_1 + a/e_2} \Leftrightarrow \\
&= \mathsf{acc}(T)^{\frac{be_2}{e_1 e_2} + \frac{ae_1}{e_1 e_2}} \Leftrightarrow \\
&= \mathsf{acc}(T)^{\frac{be_2 + ae_1}{e_1 e_2}} \Leftrightarrow \\
&= \mathsf{acc}(T)^{\frac{1}{e_1 e_2}}
\end{aligned}$$

Note that although aggregating proofs saves communication, the verification time remains as slow as verifying all the individual witnesses. For example, in a hidden-order group, the $\pi^{e_1 \cdot e_2}$ exponentiation is as slow as the individual $\pi_1^{e_1}$ and $\pi_2^{e_2}$ exponentiations since $e_1 \cdot e_2$ cannot be "reduced" in the exponent. However, as pointed out in [31], the verification time can be sped up using a proof of exponentiation (see Section 4.2.1), although at the cost of a higher aggregation time.

Divide-and-Conquer Trick for Precomputing Membership Witnesses Fast. Even though computing a single membership witness takes $\Theta(n)$ time in an RSA accumulator, computing all n membership witnesses can be done in $\Theta(n \log n)$ time [184]. Without loss of generality, assume $n = 2^m$ for some $m > 0$. The algorithm's starting input is the generator g (i.e., the accumulator of the empty set) and the set T of n elements for which witnesses must

be precomputed. Let T_L and T_R of size $n/2$ each denote the left and right "halves" of T. The algorithm computes one RSA accumulator a_L over T_L and another accumulator a_R over T_R in $\Theta(n)$ time. Then, the algorithm calls itself recursively: first with a_R and T_L as input, and second with a_L and T_R as input. As shown in Figure 4-1, this recursive computation produces a membership witness for each element e_i.

Subset and Disjointness Witnesses. Boneh et al. [31] introduced new techniques for batching and aggregating (non)membership witnesses in RSA accumulators. Although they introduce *union witnesses*, which can be easily modified into a subset witness, they do not explicitly introduce disjointness witnesses. In Section 5.4.1, we show how to use their techniques to obtain disjointness witnesses and a more efficient subset witness.

4.4 (Un)trusted Setup for Hidden-Order Groups

There are two efficient ways of instantiating a hidden-order group $\mathbb{G}_?$. The first is to pick an RSA modulus $N = pq$ as the product of two large primes p, q and set $\mathbb{G}_?$ to be $\mathbb{Z}_N^* = \{a \mid \gcd(a, N) = 1\}$ or a subgroup of $\mathbb{Z}_N^*$ such as $\mathbb{QR}_N$. The main drawback of this approach is the party who picks p and q knows the order $\varphi(N) = (p-1)(q-1)$ of the group. Thus, to avoid a trusted setup, a distributed key generation (DKG) protocol should be run amongst n players that jointly pick such an N without learning the factorization [33]. Recently, efficient DKG protocols have been proposed for the $n = 2$ setting [93,115]. Protocols that are run by $n > 2$ parties also exist but are less efficient and do not scale to a large n [70].

The second way is to work with *class groups of imaginary quadratic order* [41,42]. Unlike $\mathbb{Z}_N^*$, a class group can be picked at random without a trusted setup. It is believed that determining the order of such a group is computationally infeasible. Although class groups are relatively new, they have recently received increased interest [10,30,31,68,144,177,209].

Chapter 5

Append-only Authenticated Sets (AAS)

5.1 Overview

We begin by introducing a new primitive called an *append-only authenticated set* (AAS). An AAS is just a universal accumulator [139] that supports subset witnesses and is fork-consistent. We formalize the notion of an AAS in Section 5.2 and instantiate it *efficiently* from bilinear and RSA accumulators in Section 5.3 and in Section 5.4, respectively. Then, in Chapter 6, we slightly modify our AAS constructions to obtain *append-only authenticated dictionaries* (AADs), which can be used to implement any transparency log [83]. Nonetheless, an AAS is useful in and of itself. For example, it can be used for Revocation Transparency (RT) [136] to efficiently check the revocation status of certificates.

An AAS is a set of *elements* managed by an *untrusted server* and queried by *clients*. The server is the sole author of the AAS: it can append elements on its own and/or accept elements from clients. Clients can check membership of elements in the set (see Steps 3-5 in Figure 5-1). Clients, also known as *users*, are mutually-distrusting, potentially malicious, and do not have identities (i.e., no authorization or authentication). Initially, the set starts out empty at *version* zero, with new appends increasing its size and version by one. Importantly, once an element has been appended to the set, it remains there forever: an adversary cannot remove nor change the element. After each append, the server signs and publishes a new, small-sized *digest* of the updated set (e.g., Step 2).

Clients periodically update their view of the set by requesting a new digest from the server (e.g., Step 6 and 7). The new digest could be for an arbitrary version $j > i$, where i is the previous version of the set (not just for $j = i + 1$). Importantly, clients always ensure the set remains *append-only* by verifying an *append-only proof* $\pi_{i,j}$ between the old and new digest (e.g., Step 8). This way, clients can be certain the malicious server has not removed any elements from the set. Clients will not necessarily have the latest digest of the set. Finally, clients securely check if an element k is in the set via a *(non)membership proof* (e.g., Steps 3-5 in Figure 5-1).

A malicious server can *fork* clients' views [140], preventing them from seeing each other's appends. To deal with this, clients maintain a *fork-consistent* view [140, 141] of the set by

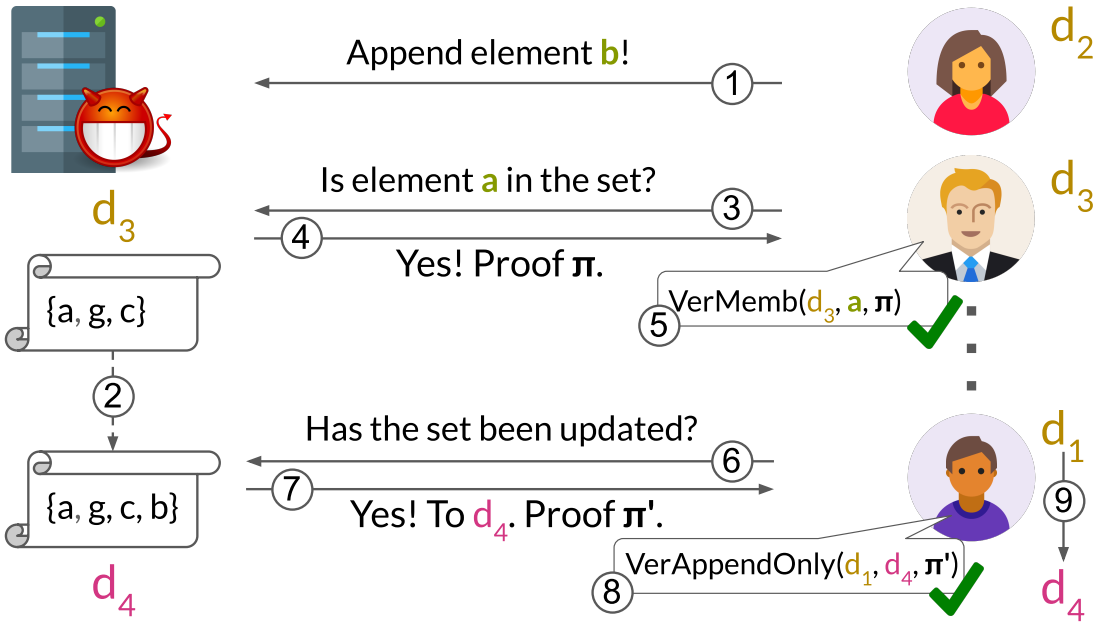


Figure 5-1: Our model: a single malicious *server* manages a *set* and many *clients* query the set. Clients will not necessarily have the digest of the latest set. The clients can (1) append a new element to the set, (2) query for an element and (3) ask for an updated digest of the set.

checking append-only proofs. As a consequence, if the server ever withholds an append from one client, that client's digest will forever diverge from other clients' digests. To detect such *forks*, clients can *gossip* [62, 67, 197, 200] with one another about their digests. This is crucial for security in transparency logs.

This model is the same as in history trees (HTs) [64], assuming only a gossip channel and no trusted third parties. It also arises in encrypted web applications [87, 124, 178], Certificate Transparency (CT) [137] and software transparency schemes [86, 162]. Unlike the 2- and 3-party models [12, 169, 180], there is no *trusted source* that signs appends in this model. A trusted source trivially solves the AAS/AAD problem as it can simply vouch for the data structure's append-only property with a digital signature. Unfortunately, this kind of solution is useless for transparency logs [137, 147, 183], which lack trusted parties.

5.2 Definitions

5.2.1 Server-side API

The untrusted server implements:

$\mathsf{Setup}(1^\lambda, B) \to \mathsf{pp}, \mathsf{VK}$. Randomized algorithm that returns public parameters pp used by the server and a *verification key* VK used by clients. Here, λ is a security parameter and B is an upper-bound on the number of elements n in the set (i.e., $n \leq B$).

$\mathsf{Append}(\mathsf{pp}, \mathcal{S}_i, d_i, k) \to \mathcal{S}_{i+1}, d_{i+1}$. Deterministic algorithm that appends a new element k to the version i set, creating a new version $i+1$ set. Succeeds only if the set is not full (i.e.,

$i+1 \leq B$). Returns the new authenticated set $\mathcal{S}_{i+1}$ and its digest d_{i+1}.

$\mathsf{ProveMemb}(\mathsf{pp}, \mathcal{S}_i, k) \rightarrow b, \pi$. Deterministic algorithm that proves (non)membership for element k. When k is in the set, generates a membership proof π and sets $b = 1$. Otherwise, generates a non-membership proof π and sets $b = 0$.

$\mathsf{ProveAppendOnly}(\mathsf{pp}, \mathcal{S}_i, \mathcal{S}_j) \rightarrow \pi_{i,j}$. Deterministic algorithm that proves $\mathcal{S}_i \subseteq \mathcal{S}_j$. In other words, generates an *append-only proof* $\pi_{i,j}$ that all elements in $\mathcal{S}_i$ are also present in $\mathcal{S}_j$. Importantly, a malicious server who removed elements from $\mathcal{S}_j$ that were present in $\mathcal{S}_i$ cannot construct a valid append-only proof.

5.2.2 Client-side API

Clients implement:

$\mathsf{VerMemb}(\mathsf{VK}, d_i, k, b, \pi) \rightarrow \{T, F\}$. Deterministic algorithm that verifies proofs returned by $\mathsf{ProveMemb}(\cdot)$ against the digest d_i. When $b = 1$, verifies k is in the set via the membership proof π. When $b = 0$, verifies k is *not* in the set via the non-membership proof π. (We formalize security in Section 5.2.3.)

$\mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_{i,j}) \rightarrow \{T, F\}$. Deterministic algorithm that ensures a set remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the set with digest d_i is a subset of the set with digest d_j. Also, verifies that d_i and d_j are digests of sets at version i and j, respectively, enforcing fork consistency.

Using the API. To use an AAS scheme, first, public parameters need to be computed using a call to $\mathsf{Setup}(\cdot)$. If the AAS scheme is trapdoored, a trusted party runs $\mathsf{Setup}(\cdot)$ and forgets the trapdoor. (This can be securely implemented via a multi-party computation protocol [106].) Once computed, the parameters can be reused by different servers for different append-only sets. $\mathsf{Setup}(\cdot)$ also returns a *public* verification key VK to all clients.

Then, the server broadcasts the initial digest d_0 of the empty set $\mathcal{S}_0$ to its many clients. Clients can concurrently start appending elements using $\mathsf{Append}(\cdot)$ calls. If the server is honest, it serializes $\mathsf{Append}(\cdot)$ calls. Eventually, the server returns a new digest d_i to clients along with an append-only proof $\pi_{0,i}$ computed using $\mathsf{ProveAppendOnly}(\cdot)$. Some clients might be offline but eventually they will receive either d_i or a newer $d_j, j > i$. Importantly, whenever clients transition from version i to j, they check an append-only proof $\pi_{i,j}$ using $\mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_{i,j})$.

Clients can look up elements in the set. The server proves (non)membership of an element using $\mathsf{ProveMemb}(\cdot)$. A client verifies the proof using $\mathsf{VerMemb}(\cdot)$ against their digest. As more elements are added by clients, the server continues to publish a new digest d_j and can prove it is a superset of any previous digest d_i using $\mathsf{ProveAppendOnly}(\cdot)$.

5.2.3 Correctness and Security

We first introduce some helpful notation for our correctness definitions. Consider an ordered sequence of n appends $(k_i)_{i\in[n]}$. Let $\mathcal{S}', d' \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{S}, d, (k_i)_{i\in[n]})$ denote a sequence of $\mathsf{Append}(\cdot)$ calls arbitrarily interleaved with other $\mathsf{ProveMemb}(\cdot)$ and $\mathsf{ProveAppendOnly}(\cdot)$ calls

such that $\mathcal{S}', d' \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{S}_{n-1}, d_{n-1}, k_n)$, $\mathcal{S}_{n-1}, d_{n-1} \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{S}_{n-2}, d_{n-2}, k_{n-1})$, $\ldots$, $\mathcal{S}_1, d_1 \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{S}, d, k_1)$. Finally, let $\mathcal{S}_0$ denote an empty AAS with empty digest d_0.

Definition 5.2.1 (Append-only Authenticated Set)**.** (Setup, Append, $\mathsf{ProveMemb}$, $\mathsf{ProveAppendOnly}$, $\mathsf{VerMemb}$, $\mathsf{VerAppendOnly}$) is a secure append-only authenticated set (AAS) if $\forall$ upper-bounds $B = \mathsf{poly}(\lambda)$ it satisfies the following properties:

Definition 5.2.2 (AAS Membership Correctness)**.** $\forall n \leq B$, $\forall$ ordered sequences of appends $(k_i)_{i\in[n]}$, for all elements k, where $b = 1$ if $k \in (k_i)_{i\in[n]}$ and $b = 0$ otherwise,

$$\Pr\left[\begin{array}{c}(\mathsf{pp}, \mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B),\\ (\mathcal{S}, d) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{S}_0, d_0, (k_i)_{i\in[n]}),\\ (b', \pi) \leftarrow \mathsf{ProveMemb}(\mathsf{pp}, \mathcal{S}, k) :\\ b = b' \wedge \mathsf{VerMemb}(\mathsf{VK}, d, k, b, \pi) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Observation: Note that this definition compares the returned bit b' with the "ground truth" in $(k_i)_{i\in[n]}$ and thus provides membership correctness. Also, it handles non-membership correctness since b' can be zero. Finally, the definition handles all possible orders of appending elements.

Definition 5.2.3 (AAS Membership Security)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp}, \mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B),\\ (d, k, \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{VK}) :\\ \mathsf{VerMemb}(\mathsf{VK}, d, k, 0, \pi,) = T \wedge\\ \mathsf{VerMemb}(\mathsf{VK}, d, k, 1, \pi',) = T\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation (1): This definition captures the lack of any "ground truth" about what was inserted in the set, since there is no trusted source in our model. Nonetheless, given a fixed digest d, our definition prevents *all* equivocation attacks about the membership of an element in the set.
Observation (2): These definitions imply *collision-resistance*: i.e., different sets cannot have the same digest. To see this, suppose you have two different sets with the same digest. Since the sets differ, there exists an element x that is in the first set but not the second set (without loss of generality). Then, membership correctness guarantees you can create a valid membership proof for x w.r.t. the first set and a non-membership proof for x w.r.t. the second set. As a result, we will have both a membership and a non-membership proof for x that verifies w.r.t. the same digest. This breaks membership security and leads to a contradiction.

Definition 5.2.4 (AAS Append-only Correctness)**.** $\forall n \leq B$, $\forall m < n$, $\forall$ sequences of appends $(k_i)_{i\in[n]}$ where $n \geq 2$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp}, \mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B)\\ (\mathcal{S}_m, d_m) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{S}_0, d_0, (k_i)_{i\in[m]}),\\ (\mathcal{S}_n, d_n) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{S}_m, d_m, (k_i)_{i\in[m+1,n]}),\\ \pi \leftarrow \mathsf{ProveAppendOnly}(\mathsf{pp}, \mathcal{S}_m, \mathcal{S}_n) :\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_m, m, d_n, n, \pi) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 5.2.5 (AAS Append-only Security)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B)\\ (d_i, d_j, i<j, \pi_a, k, \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{VK}) :\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_a) = T \wedge\\ \mathsf{VerMemb}(\mathsf{VK}, d_i, k, 1, \pi) = T \wedge\\ \mathsf{VerMemb}(\mathsf{VK}, d_j, k, 0, \pi') = T\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This definition ensures that elements can only be added to an AAS.

Definition 5.2.6 (Fork Consistency)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B)\\ (d_i \neq d_i', d_j, i<j, \pi_i, \pi_i') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{VK}) :\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_i) = T \wedge\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_i', i, d_j, j, \pi_i') = T\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This is our own version of fork consistency that captures what is known in the literature about fork consistency [64, 141]. Specifically, it allows a server to fork the set at version i by presenting two different digests d_i and d_i' and prevents the server from forging append-only proofs that "join" the two forks into some common digest d_j at a later version j.

5.3 AAS from Bilinear Accumulators

This section presents our bilinear accumulator-based AAS construction. We give a more formal, algorithmic description of this construction in Section 5.3.5. We prove this construction secure under the q-SBDH and q-PKE assumptions in Section 5.3.6.1. Finally, in Section 5.4, we present our RSA accumulator-based AAS.

As mentioned in Section 3.1, a bilinear accumulator over n elements is already an AAS, but with two caveats: it is not fork-consistent (see Definition 5.2.6) and it is computationally inefficient. Specifically, proving (non)membership in a bilinear accumulator requires an $O(n)$ time polynomial division. As a consequence, precomputing all n membership proofs (naively) takes $O(n^2)$ time, which is prohibitive for most use cases. Even worse, for non-membership, one must precompute proofs for all possible missing elements, of which there are exponentially many (in the security parameter λ). Therefore, we need new techniques to achieve our desired polylogarithmic time complexity for computing both types of proofs in our AAS.

5.3.1 Precomputing Membership with Communion Trees (CTs)

Our first technique is to deploy the bilinear accumulator in a tree structure, as follows. We start with the elements e_i as leaves of a binary tree (see Figure 5-2). Specifically, each leaf will store an accumulator over the singleton set $\{e_i\}$. Every internal node in the tree will then store an accumulator over the union of the sets corresponding to its two children. For example, the parent node of the two leaves corresponding to $\{e_i\}$ and $\{e_{i+1}\}$ stores the

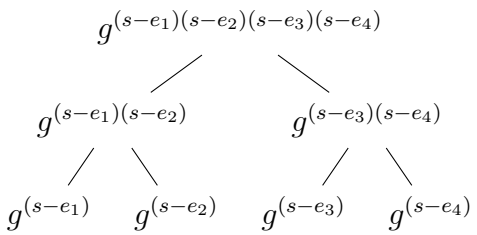


Figure 5-2: A Communion Tree (CT) over the set $\{e_1, e_2, e_3, e_4\}$. The leaves store bilinear accumulators over the individual elements. Every non-leaf node stores a bilinear accumulator over all elements from its subtree's leaves.

accumulator of the set $\{e_i, e_{i+1}\}$. In this way, the root is the accumulator over the full set $S = \{e_1, \ldots, e_n\}$ (see Figure 5-2).

We stress that all accumulators in the tree use the same public parameters. The time to compute all the accumulators in the tree is $T(n) = 2T(n/2) + O(n \log n) = O(n \log^2 n)$ where $O(n \log n)$ is the time to multiply the characteristic polynomials of two sets of size n in the tree. We call the resulting structure a *Comm(itment) Union Tree* or *Communion Tree* (CT) over set S, since every node in the tree is a commitment to the union of that node's childrens' committed sets. A similar technique of "unioning-and-then-accumulating" sets in a tree first appeared in [49].

A membership proof for element e_i will leverage the fact that sets along the path from e_i's leaf to the root of the Communion Tree are subsets of each other. The proof will consist of a sequence of *subset witnesses* that validate this (computed as explained in Section 4.3.1). Specifically, the proof contains the accumulators along the path from e_i's leaf to the root, as well as the accumulators of all sibling nodes along this path (see Figure 5-2). The client verifies all these subset witnesses, starting from the singleton set $\{e_i\}$ in the leaf. This convinces him that e_i is contained in the parent's accumulated set, which in turn is contained in its parent's accumulated set and so on, until the root.

Our CT approach gives us membership proofs of logarithmic size and thus logarithmic verification time. Importantly, computing a Communion Tree in $O(n \log^2 n)$ time implicitly computes all membership proofs "for free"! In contrast, building a standard billinear accumulator over S would yield constant-size proofs but in $O(n^2)$ time for all n proofs. Unfortunately, CTs cannot (yet) precompute non-membership proofs, which we address next.

5.3.2 Precomputing Non-membership by Accumulating Prefixes

The Communion Tree (CT) technique can efficiently precompute all n membership proofs. For non-membership though, we have to somehow precompute proofs for an exponential number of elements in the security parameter λ. (Recall that an element is just a number $e \in \mathbb{F}_p$ where $p \approx 2^{2\lambda}$.) For this, we build upon old ideas in computer science.

Specifically, we represent the set of elements as binary strings of length 2λ bits. Thus, the set can be viewed as a *trie* (see Figure 5-3a). As a consequence, when an element is not in the set, one of the prefixes of its binary string will not be in the trie. Our key

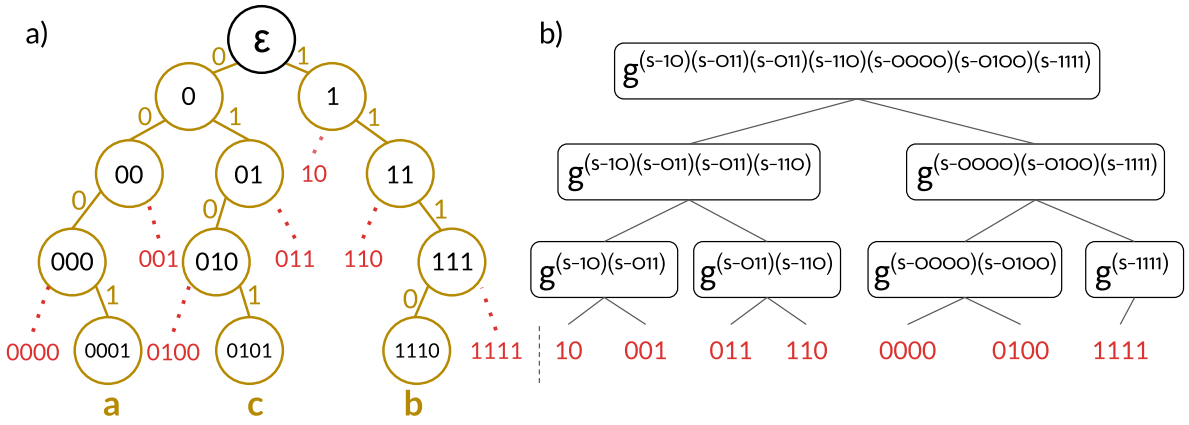


Figure 5-3: On the left side, we depict a trie over set $S = \{a, b, c\}$. Each element is mapped to a unique path of length 4 in the trie. (Here, $\lambda = 2$.) Nodes that are not in the trie but are at its *frontier* are depicted in **red**. On the right side, we depict a *Frontier Communion Tree* (FCT) corresponding to the set S. To prove that an element is not in S, we prove one of its prefixes is in the FCT.

observation is that precomputing all proofs for such missing prefixes in effect precomputes all non-membership proofs. This "missing prefixes" technique is also used in Micali et al.'s zero-knowledge sets [152]. We introduce it gradually below.

5.3.2.1 Prefix Communion Trees (PCTs)

To efficiently precompute non-membership proofs, we slightly modify our CT construction into a Prefix Communion Tree (PCT). As before, a parent's set is the union of its children's sets, but the key difference is that leaves will no longer store an individual element e_i but will store all prefixes $\mathsf{P}(e_i)$ of its binary representation. We assume this representation is 2λ bits (or is made so using a CRHF) and can be mapped to an element in $\mathbb{F}_p$ (which is also of size $\approx 2\lambda$ bits) and thus can be accumulated.

For example, a leaf that previously stored element e_1 with binary representation 0001, will now store the set $\mathsf{P}(e_1) = \{\varepsilon, 0, 00, 000, 0001\}$ (i.e., all the prefixes of the binary representation of e_1, including the empty string ε). Also, for any set $S = \{e_1, \ldots, e_n\}$, we define its *prefix set* as $\mathsf{P}(S) = \mathsf{P}(e_1) \cup \cdots \cup \mathsf{P}(e_n)$. For example, let $S = \{a = 0001, b = 0101, c = 1110\}$. Then, the root of S's Prefix Communion Tree will contain a *prefix accumulator* over $\mathsf{P}(S) = \mathsf{P}(a) \cup \mathsf{P}(b) \cup \mathsf{P}(c) = \{\varepsilon, 0, 1, 00, 01, 11, 000, 010, 111, 0001, 0101, 1110\}$. We refer to accumulators in PCT nodes as prefix accumulators since they accumulate prefixes of elements, rather than elements themselves.

The time to build a PCT for S is $O(\lambda n \log^2 n)$ since there are $O(\lambda n)$ prefixes across all leaves. Note that membership proofs in a PCT are the same as in CTs, with a minor change. The internal nodes of the tree still store accumulators over the union of their children. However, the children now have common prefixes, which will only appear once in the parent. For example, two children sets have the empty string ε while their parent set only has ε once (because of the union). As a result, it is no longer the case that multiplying the characteristic polynomials of the children gives us the parent's polynomial. Therefore, we can no longer rely

on the siblings as subset witnesses: we have to explicitly compute subset witnesses for each child w.r.t. its parent. We stress that this does not affect the asymptotic time complexity of computing the PCT. As before, the client starts the proof verification from the leaf, which now stores a prefix set $\mathsf{P}(e_i)$ rather than a singleton set $\{e_i\}$.

5.3.2.2 Frontier Communion Tree (FCTs)

But how does a PCT help with precomputing non-membership proofs for any element $e' \notin S$? First, note that to prove $e' \notin S$ it suffices to show that *any one prefix ρ of e' is not contained in* $\mathsf{P}(S)$. Second, note that there might exist other elements e'' who share ρ as a prefix. As a result, the non-membership proof for e' could be "reused" as a non-membership proof for e''.

This is best illustrated in Figure 5-3a using our previous example where $S = \{a, b, c\}$. Consider elements $d = \underline{0111}1$ and $f = \underline{0110}$ that are not in S. To prove non-membership for either element, it suffices to prove the same statement: $\underline{011} \notin \mathsf{P}(S)$. Thus, if we can identify all such shared prefixes, we can use them to prove the non-membership of (exponentially) many elements.

First, note that prefix accumulator of a set S is just a *trie*, as depicted in Figure 5-3a. The key idea is to keep track of the prefixes at the "frontier" of the trie, depicted in **red** in Figure 5-3a. Informally, these *frontier prefixes* are prefixes that are *not* in the trie but have a *parent* in the trie (e.g., 011 is not in the trie but its parent 01 is). We immediately notice that to prove non-membership of any element, it suffices to prove non-membership of one of these frontier prefixes! In other words, elements that are not in S will have one of these as a prefix. We can formally define the *frontier* of S as:

$$F(S) = \{\rho \in \{0,1\}^{\leq 2\lambda} : \rho \notin \mathsf{P}(S) \wedge \mathsf{parent}(\rho) \in \mathsf{P}(S)\},$$

where $\mathsf{parent}(\rho)$ is ρ without its last bit (e.g., $\mathsf{parent}(011) = 01$). Note that the size of $F(S)$ is $O(\lambda n)$, proportionate to $\mathsf{P}(S)$.

Most importantly, from the way $\mathsf{P}(S)$ and $F(S)$ are defined, for any element e', it holds that $e' \notin S$ if, and only if, some prefix of e' is in $F(S)$. Therefore, proving non-membership of e' boils down to proving two statements: (i) some prefix of e' belongs to $F(S)$, and (ii) $\mathsf{P}(S) \cap F(S) = \varnothing$. We stress that the latter is necessary as a malicious server may try to craft $F(S)$ in a false way (e.g., by adding some prefixes both in $\mathsf{P}(S)$ and in $F(S)$).

To prove (i), we build a Communion Tree over $F(S)$ which gives us precomputed membership proofs for all $\rho \in F(S)$ (see Figure 5-3b). We refer to this tree as the *Frontier Communion Tree (FCT)* for set S, to the proofs as *frontier proofs*, and to the accumulators in the tree as *frontier accumulators*. To prove (ii), we compute a *disjointness witness* between sets $\mathsf{P}(S)$ and $F(S)$, as described in Section 2.5 (i.e., between the root prefix accumulator and the root frontier accumulator). The time to build a FCT for S is $O(\lambda n \log^2 n)$ since $F(S)$ has $O(\lambda n)$ elements. The disjointness witness can also be computed in $O(\lambda n \log^2 n)$ time.

5.3.3 From Static to Dynamic AAS

Combining all the above techniques, we obtain a *static* AAS that does *not yet* support updates efficiently (nor append-only proofs). This construction consists of: (a) a PCT for S,

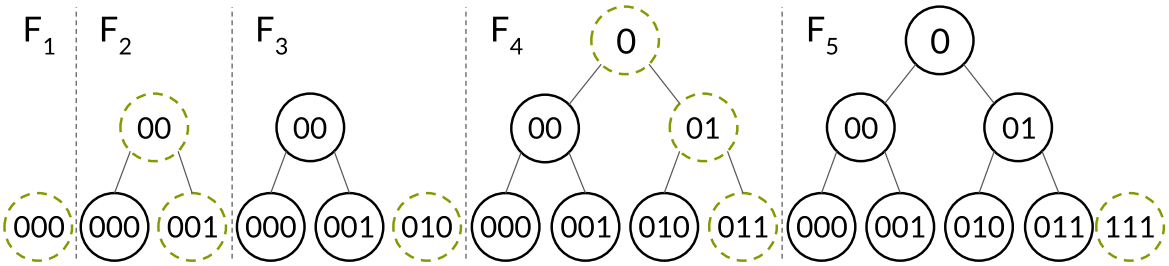


Figure 5-4: A forest starting empty and going through a sequence of five appends. A forest only has trees of exact size 2^j for distinct j's. A forest of n leaves has *at most* $\log n$ trees.

(b) a FCT for S, and (c) a disjointness witness for $\mathsf{P}(S)$ and $F(S)$ (i.e., between the root prefix and frontier accumulators). The height of the PCT is $O(\log n)$ and the height of the FCT is $O(\log(\lambda n))$ so the size and verification time of a (non)membership proof is $O(\log n)$. The digest is just the root prefix accumulator of the PCT.

5.3.3.1 Appending Efficiently in Polylogarithmic Time

Our AAS should *efficiently* support appending new elements to S. The main challenge here is that updating the PCT and FCT as well as the disjointness witness after each update is very expensive (i.e., at least linear time in their size). To address this we use a classic "amortization" trick from Overmars [165] also used in [181]. Specifically, our AAS will consist not of one PCT for the entire set S, but of a *forest* of PCTs and their corresponding FCTs. The idea is to maintain a partitioning of S with $1 + \lfloor \log |S| \rfloor$ disjoint subsets, each of a distinct size 2^i.

Initially, we start with no elements in the AAS. When the first element e_1 is appended, we build its PCT over the set $\{e_1\}$, its FCT and a disjointness witness. Together, these make up the *CT-pair* of a set S (in this case, $S = \{e_1\}$). When the second element e_2 is appended, we "merge": we build a CT-pair over $\{e_1, e_2\}$. We define the *size of a CT pair* as the size of its set S (in this case, the size is 2 since $S = \{e_1, e_2\}$). The rule is we always merge equal-sized CT-pairs. When e_3 is appended, we cannot merge it because there's no other CT-pair of size 1. Instead, we create a CT-pair over $\{e_3\}$.

In general, after $2^\ell - 1$ appends, we end up with ℓ separate CT-pairs corresponding to sets of elements $S_1, \ldots, S_\ell$. The final set is $S = \bigcup_{j=1}^{\ell} S_j$ where $|S_j| = 2^j$. The evolution of such a forest is depicted in Figure 5-4 and the final data structure can be seen in Figure 5-5. In Section 5.3.4, we show this approach gives us an $O(\lambda n \log^3 n)$ *amortized* append time. Fortunately, generic *de-amortization* techniques [165, 166] can be used to obtain an $O(\lambda n \log^3 n)$ *worst-case* append time.

The downside of our amortized approach is that proving non-membership becomes slightly more expensive than in the static AAS data structure from above. Specifically, now the server needs to prove non-membership in each CT-pair separately, requiring an $O(\log n)$ frontier proof in each of the $O(\log n)$ FCTs. This increases the non-membership proof size to $O(\log^2 n)$. On a good note, membership proofs remain unaffected: the server just sends a path to a leaf in *one* of the PCTs where the element is found. Finally, the AAS digest is set to the root prefix accumulators of all PCTs and has size $O(\log n)$. We analyze the complexity

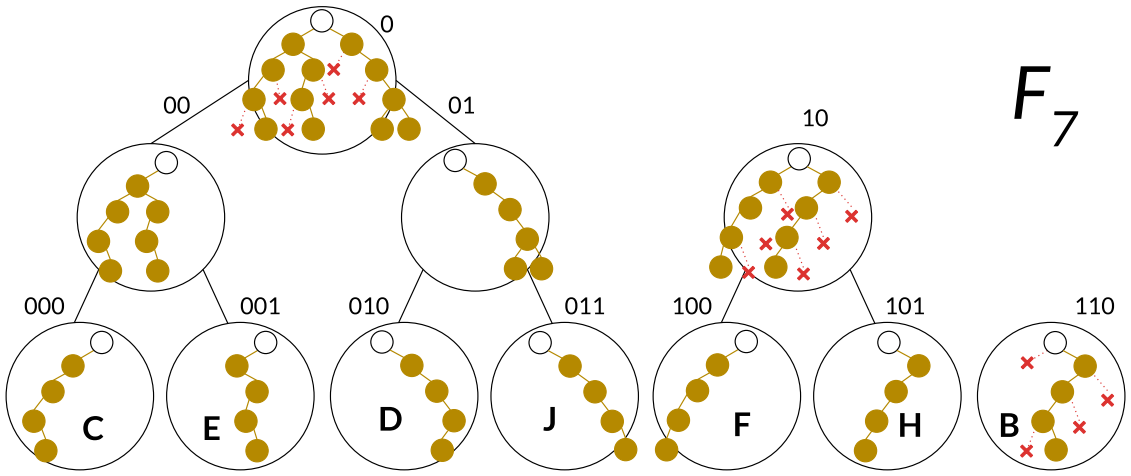


Figure 5-5: A dynamic AAS with $\lambda = 2$ for set $\{B, C, D, E, F, H, J\}$. Our AAS is a forest of PCTs with corresponding FCTs. Each node stores a prefix accumulator (and subset witness), depicted as a trie, in yellow. Root nodes store an FCT, depicted as the missing red nodes.

of our AAS in Section 5.3.4.

5.3.3.2 Logarithmic-sized Append-only Proofs

Our append-only proofs are similar to the ones in history trees [64]. An append-only proof must relate the root prefix accumulator(s) in the old AAS to the root prefix accumulator(s) in the new AAS. We'll refer to these as "old roots" and "new roots", respectively. Specifically, it must show that every old root either (i) became a new root or (ii) has a path to a new root with valid accumulator subset witnesses at every level. Such a path is verified by checking the subset witnesses between every child and its parent, exactly as in a membership proof. At the same time, note that there might be new roots that are neither old roots nor have paths to old roots (e.g., root 111 in F_5 from Figure 5-4). The proof simply ignores such roots since they securely add new elements to the set. To summarize, the append-only proof guarantees that each old root (1) has a valid subset path to a new root or (2) became a new root.

Ensuring Fork Consistency. For gossip protocols to work [62, 67], our AAS must be fork-consistent. Interestingly, append-only proofs do not imply fork consistency. For example, consider a server who computes an AAS for set $\{e_1\}$ and another one for the set $\{e_2\}$. The server gives the first set's digest to user A and the second digest to user B. Afterwards, he appends e_2 to the first set and e_1 to the second one, which "joins" the two sets into a common set $\{e_1, e_2\}$. The append-only property was not violated (as the two users can deduce independently) but fork consistency has been: the two users had diverging views that were subsequently merged.

To avoid this, we will "Merkle-ize" each PCT using a CRHF $\mathcal{H}$ in the standard manner (i.e., a node hashes its prefix accumulator and its two children's hashes). Our AAS digest is now set to the Merkle roots of all PCTs, which implicitly commit to all root prefix accumulators in the PCTs. As a result, after merging PCTs for elements e_1 and e_2, the Merkle root of

the merged PCT will differ based on how appends were ordered: (e_1, e_2), or (e_2, e_1). Thus, violating fork consistency becomes as hard as finding a collision in $\mathcal{H}$. (We prove this in Section 5.3.6.3.)

5.3.4 Asymptotic Analysis

Suppose we have a *worst-case* AAS with $n = 2^i - 1$ elements.

Append Time. First, let us analyze the time to merge two size-n CT-pairs for two sets S_1 and S_2 into a size-$2n$ CT-pair for their union $S = S_1 \cup S_2$. To compute S's PCT, we need to (i) compute its prefix accumulator, (ii) set its children to the "old" prefix accumulators of S_1 and S_2 and (iii) compute subset witnesses for $S_1 \subset S$ and for $S_2 \subset S$. Since $|S_1| = |S_2| = n$, operations (i), (ii) and (iii) take $O(\lambda n \log^2 n)$ time. Finally, we can compute S's FCT from scratch in $O(\lambda n \log^2 n)$ time.

Now, consider the time $T(n)$ to create an AAS over a set S with $n = 2^\ell$ elements (without loss of generality). Then, $T(n)$ can be broken into:

- The time to create a CT-pair over the children of S of size $n/2$ (i.e., $2T(n/2)$).
- The time to merge these two CT-pairs, including computing subset witnesses (discussed above)
- The time to compute the FCT of S (discussed above).

More formally, $T(n) = 2T(n/2) + O(\lambda n \log^2 n)$ which simplifies to $T(n) = O(\lambda n \log^3 n)$ time for n appends. Thus, the *amortized* time for one append is $O(\lambda \log^3 n)$ and can be de-amortized into *worst-case* time using generic techniques [165, 166].

Space. The space is dominated by the FCTs, which take up $O(\lambda n/2)+O(\lambda n/4)+\cdots+O(1) = O(\lambda n)$ space. (When accounting just for the prefix accumulators, PCTs only take up $O(n)$ space.)

Digest Size. The digest is $O(\log n)$-sized where n is the size of the set. We can make the digest constant-sized by hashing all Merkle roots together. Then, we can include the Merkle roots as part of our append-only and lookup proofs, without increasing our asymptotic proof sizes.

Membership Proof Size. Suppose an element e is in some PCT of the AAS . To prove membership of e, we show a path from e's leaf in the PCT to the PCT's root prefix accumulator consisting of constant-sized subset witnesses at every node. Since the largest PCT in the forest has height $\log(n/2)$, the membership proof is $O(\log n)$-sized.

Non-membership Proof Size. To prove non-membership of an element e, we show a frontier proof for a prefix of e in every FCT in the forest. The largest FCT has $O(\lambda n)$ nodes so frontier proofs are $O(\log(\lambda n))$-sized. Because there are $O(\log n)$ FCTs, all the frontier proofs are $O(\log n \log(\lambda n)) = O(\log^2 n)$-sized.

Append-only Proof Size. Our append-only proof is $O(\log n)$-sized. This is because, once we exclude common roots between the old and new digest, our proof consists of paths from each old root in the old forest up to a single new root in the new forest. Because the old roots are roots of adjacent trees in the old forest, there will be a single $O(\log n)$-sized Merkle path connecting the old roots to the new root. In other words, our append-only proofs are similar to the append-only proofs from history trees [64].

Public Parameters. The server needs q-SDH public parameters, where $q = \Theta(\lambda n)$. This is because, for an AAS of size n, it needs to build a trie of height 2λ over the n keys. In other words, the server has to accumulate $< (2\lambda + 1)n$ prefixes which requires $((2\lambda + 1)n)$-SDH parameters. When verifying (non)membership proofs, a client must reconstruct a leaf prefix accumulator, which accumulates $2\lambda + 1$ prefixes. Thus, they only need $(2\lambda + 1)$-SDH public parameters (see Algorithm 1).

5.3.5 Algorithms

Here, we give detailed algorithms that implement our amortized, dynamic AAS from Section 5.3. Recall that our AAS is just a forest of PCTs with corresponding FCTs. Importantly, recall that our dynamic AAS is a forest that grows over time, as depicted in Figure 5-4. In particular, observe that each forest node has a prefix accumulator associated with it, while root nodes in the forest have FCTs associated with them. Our algorithms described below operate on this forest, adding new leaves, merging nodes in the forest and computing FCTs in the roots.

An important detail is that a bilinear accumulators $g^{\mathcal{C}_A(s)}$ for a set A is often accompanied by an "extractable" counterpart $g^{\tau\mathcal{C}_A(s)}$. Here s is the q-SDH trapdoor and τ denotes another trapdoor. The extractable counterpart is necessary to prove security of our (non)membership proofs and our append-only proofs under q-PKE (see Section 5.3.6).

Trees Notation. The $|$ symbol denotes string concatenation. A *tree* is a set of nodes denoted by binary strings in a canonical way. The root of a tree is denoted by the empty string ε and the left and right children of a node w are denoted by $w|0$ and $w|1$, respectively. If $b \in \{0, 1\}$, then the sibling of $w = v|b$ is denoted by $\mathsf{sibling}(w) = v|\overline{b}$, where $\overline{b} = 1 - b$. A *path* from one node v to its ancestor node w is denoted by $\mathsf{path}[v, w] = \{u_1 = v, u_2 = \mathsf{parent}(u_1), \ldots, u_\ell = \mathsf{parent}(u_{\ell-1}) = w\}$. The parent node of $v = w|b$ is denoted by $\mathsf{parent}(v) = \mathsf{parent}(w|b) = w$. We also use $\mathsf{path}[v, w) = \mathsf{path}[v, w] - \{w\}$.

Forest Notation. Let F_i denote a forest of $\leq B$ leaves that only has i leaves in it. (For example, Figure 5-4 depicts a forest with $B = 8$ growing from one leaf to five leaves.) Intuitively, a forest is a set of trees where each tree's size is a *unique* power of two (e.g., see F_5 in Figure 5-4). The unique tree sizes are maintained by constantly merging trees of the same size. Let $\mathsf{bin}^B(x)$ denote the $\lceil \log B \rceil$-bit binary expansion of a number x (e.g., $\mathsf{bin}^{14}(6) = 0110$). (Note that $\mathsf{bin}^1(x) = \varepsilon, \forall x$ because $\lceil \log 1 \rceil = 0$.) In our AAS, $\mathsf{bin}^B(i)$ denotes the ith inserted leaf, where i starts at 0 (e.g., see leaves 000 through 111 in F_5 in Figure 5-4). Let $\mathsf{roots}(F_i)$

denote all the roots of all the trees in the forest (e.g., $\mathsf{roots}(F_5) = \{0, 111\}$ in Figure 5-4). Let $\mathsf{leaves}(F_i)$ denote all the leaves in the forest (e.g., $\mathsf{leaves}(F_3) = \{000, 001, 010\}$ in Figure 5-4).

AAS Notation. Our algorithms use **assert** to ensure a condition is true or fail the calling function otherwise. Let $\mathrm{Dom}(f)$ be the domain of a function f. We use $f(x) = \perp$ to indicate $x \notin \mathrm{Dom}(f)$. Let $\mathcal{S}_i$ denote our AAS with i elements. Each node w in the forest stores "extractable" accumulators $\mathbf{a}_w, \hat{\mathbf{a}}_w$ of its PCT together with a Merkle hash $\mathbf{h}_w$. Internal nodes (i.e., non-roots) store a subset witness $\boldsymbol{\pi}_w$ between $\mathbf{a}_w$ and $\mathbf{a}_{\mathsf{parent}(w)}$. The digest d_i of $\mathcal{S}_i$ maps each root r to its Merkle hash $\mathbf{h}_r$. Every root r stores a disjointness witness $\boldsymbol{\psi}_r$ between its PCT and FCT. For simplicity, we assume server algorithms implicitly "parse out" the **bolded blue variables** from $\mathcal{S}_i$.

Server Algorithms. $\mathsf{Setup}(\cdot)$ generates large enough q-PKE public parameters $\mathsf{PP}_q^{\mathsf{PKE}}(g; s, \tau)$ (see Definition 2.1.4), given an upper bound B on the number of elements. Importantly, the server forgets the trapdoors s and τ used to generate the public parameters. In other words, this is a *trusted setup* phase (see Section 2.6).

Algorithm 1 Computes public parameters (trusted setup)

function $\mathsf{Setup}(1^\lambda, B) \to (\mathsf{pp}, \mathsf{VK})$ ▷ Generates q-PKE public parameters
  $\ell \leftarrow 2^{\lfloor \log B \rfloor}$ $q \leftarrow (2\lambda + 1)\ell$ $(\mathbb{G}, \mathbb{G}_T, p, g, e(\cdot,\cdot)) \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$
  $s \xleftarrow{\$} \mathbb{F}_p$ $\tau \xleftarrow{\$} \mathbb{F}_p$ $\mathsf{VK} = ((g^{s^i})_{i=0}^{2\lambda+1}, g^\tau)$
  return $(((\mathbb{G}, \mathbb{G}_T, p, g, e(\cdot,\cdot)), B, \mathsf{PP}_q^{\mathsf{PKE}}(g; s, \tau)), \mathsf{VK})$

Algorithm 2 Appends a new ith element to the AAS, $i \in [0, B-1]$

function $\mathsf{Append}(\mathsf{pp}, \mathcal{S}_i, d_i, k) \to (\mathcal{S}_{i+1}, d_{i+1})$
  $w \leftarrow \mathsf{bin}^B(i)$ $\mathbf{S}_w \leftarrow \{k\}$ ▷ Create new leaf w for element k
  $(\alpha_w, \mathbf{a}_w, \cdot) \leftarrow \mathsf{Accum}(\mathsf{P}(\mathbf{S}_w))$ $\mathbf{h}_w \leftarrow \mathcal{H}(w|\perp|\mathbf{a}_w|\perp)$
  ▷ "Merge" old PCT roots with new PCT root (recursively)
  while $\mathsf{sibling}(w) \in \mathsf{roots}(F_i)$ **do**
    $\ell \leftarrow \mathsf{sibling}(w)$ $p \leftarrow \mathsf{parent}(w)$ $\mathbf{S}_p \leftarrow \mathbf{S}_\ell \cup \mathbf{S}_w$
    $(\alpha_p, \mathbf{a}_p, \hat{\mathbf{a}}_p) \leftarrow \mathsf{Accum}(\mathsf{P}(\mathbf{S}_p))$ $\mathbf{h}_p = \mathcal{H}(p|\mathbf{h}_\ell|\mathbf{a}_p|\mathbf{h}_w)$
    $(\cdot, \boldsymbol{\pi}_\ell, \cdot) \leftarrow \mathsf{Accum}(\mathsf{P}(\mathbf{S}_p \setminus \mathbf{S}_\ell))$
    $(\cdot, \boldsymbol{\pi}_w, \cdot) \leftarrow \mathsf{Accum}(\mathsf{P}(\mathbf{S}_p \setminus \mathbf{S}_w))$ $w \leftarrow p$
  ▷ Invariant: w is a new root in F_{i+1}. Next, computes w's frontier.
  $(\phi_w, \boldsymbol{\sigma}_w) \leftarrow \mathsf{CreateFrontier}(F(\mathbf{S}_w))$
  $(y, z) \leftarrow \mathsf{ExtEuclideanAlg}(\alpha_w, \phi_w)$ $\boldsymbol{\psi}_w \leftarrow (g^{y(s)}, g^{z(s)})$
  Store updated AAS state (i.e., the **bolded blue** variables) into $\mathcal{S}_{i+1}$
  $d_{i+1}(r) \leftarrow \mathbf{h}_r, \forall r \in \mathsf{roots}(F_{i+1})$ ▷ Set new digest
  return $\mathcal{S}_{i+1}, d_{i+1}$
function $\mathsf{Accum}(T)$
  return $(\alpha, g^{\alpha(s)}, g^{\tau\alpha(s)})$ where $\alpha(x) = \prod_{w \in T} (x - \mathcal{H}_{\mathbb{F}}(w))$

$\mathsf{Append}(\cdot)$ creates a new leaf ℓ for the element k (Lines 2 to 3). It recursively merges equal-sized PCTs in the forest, as described in Section 5.3 (Lines 5 to 9). In this process, it computes subset witnesses between old PCT roots and the new PCT. Merging ends when the newly created PCT w has no equal-sized PCT to be merged with. Recall from Section 2.5 that $\mathcal{H}_{\mathbb{F}}$ maps elements to be accumulated to field elements in $\mathbb{F}_p$.

If k is in the set, $\mathsf{ProveMemb}(\cdot)$ sends a Merkle path to k's leaf in some tree with root r (Lines 3 to 5) via $\mathsf{ProvePath}(\cdot)$ (see Algorithm 3). This path contains subset witnesses between every node's accumulator and its parent node's accumulator. If k is not in the set, then $\mathsf{ProveMemb}(\cdot)$ sends frontier proofs in each FCT in the forest (Lines 6 to 8) via $\mathsf{ProveFrontier}(\cdot)$ (see Algorithm 6).

Algorithm 3 Constructs a (non)membership proof

1: **function** $\mathsf{ProveMemb}(\mathsf{pp}, \mathcal{S}_i, k) \to (b, \pi)$
2: Let $\ell \in \mathsf{leaves}(F_i)$ be the leaf where k is stored or $\perp$ if $k \notin \mathcal{S}_i$
3: **if** $k \in \mathcal{S}_i$ **then** ▷ Construct Merkle path to element
4: Let $r \in \mathsf{roots}(F_i)$ be the root of the tree where k is stored
5: $\pi \leftarrow \mathsf{ProvePath}(\mathcal{S}_i, \ell, r, \perp)$ $b \leftarrow 1$
6: **else** ▷ Prove non-membership in all FCTs
7: $\chi_r \leftarrow \mathsf{ProveFrontier}(\mathcal{S}_i, r, k), \forall r \in \mathsf{roots}(F_i)$
8: $\pi \leftarrow \mathsf{ProveRootAccs}(\mathcal{S}_i, \pi)$ $b \leftarrow 0$
9: **return** $b, (\ell, \pi, (\chi_r)_{r \in \mathsf{roots}(F_i)}, (\boldsymbol{\psi}_r)_{r \in \mathsf{roots}(F_i)})$
10: **function** $\mathsf{ProvePath}(\mathcal{S}_i, u, r, \pi) \to \pi$ ▷ Precondition: r is a root in F_i
11: $\pi(r) \leftarrow (\perp, \mathbf{a}_r, \hat{\mathbf{a}}_r, \perp)$
12: ▷ Overwrites $\pi(w)$ set by previous $\mathsf{ProvePath}$ call (if any)
13: $\pi(w) \leftarrow (\perp, \mathbf{a}_w, \hat{\mathbf{a}}_w, \boldsymbol{\pi}_w), \forall w \in \mathsf{path}[u, r)$
14: ▷ Only sets $\pi(\mathsf{sibling}(w))$ if not already set from previous $\mathsf{ProvePath}$ call!
15: **for** $w \in \mathsf{path}[u, r)$ where $\mathsf{sibling}(w) \notin \mathrm{Dom}(\pi)$ **do**
16: $\pi(\mathsf{sibling}(w)) \leftarrow (\mathbf{h}_{\mathsf{sibling}(w)}, \perp, \perp, \perp)$
17: **return** π
18: **function** $\mathsf{ProveRootAccs}(\mathcal{S}_i, \pi) \to \pi$
19: $\pi(r) \leftarrow (\perp, \mathbf{a}_r, \hat{\mathbf{a}}_r, \perp), \forall r \in \mathsf{roots}(F_i),$
20: $\pi(r|c) \leftarrow (\mathbf{h}_{r|c}, \perp, \perp, \perp), \forall r \in \mathsf{roots}(F_i), \forall c \in \{0, 1\}$

For each root r in F_i, $\mathsf{ProveAppendOnly}(\cdot)$ sends a Merkle path to an ancestor root in F_j, if any. The Merkle path contains subset witnesses between all prefix accumulators along the path. It also contains the root prefix accumulators from F_i, which the client will verify against his digest d_i.

Client Algorithms. $\mathsf{VerAppendOnly}(\cdot)$ first ensures that d_i and d_j are digests at version i and j, respectively (Lines 7 to 8). This involves checking that d_i has the expected number of roots (i.e., same as in F_i) and that the Merkle hash of each root r in d_i is computed with the correct label r as $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$. However, for simplicity of exposition, Lines 7 to 8 of $\mathsf{VerAppendOnly}(\cdot)$ leave these details out.

Before checking subset witnesses, $\mathsf{VerAppendOnly}(\cdot)$ validates the old root prefix accumulators in $\pi_{i,j}$ against the Merkle roots in d_i (Lines 11 to 13). Without this check, any old root prefix accumulator could be maliciously given in the append-only proof $\pi_{i,j}$, breaking the append-only property. Then, it checks that each root r from F_i is a subset of some root in F_j by checking subset witnesses (Line 16) via $\mathsf{VerPath}(\cdot)$ (see Algorithm 5).

$\mathsf{VerAppendOnly}(\cdot)$ enforces fork consistency implicitly when verifying Merkle hashes. For this to work, Line 15 first ensures that the Merkle paths from the old roots to the new roots are "well-formed." This means checking that the proof paths contains all the expected sibiling Merkle hashes, at the right positions, and no other Merkle hashes. If other Merkle hashes were included, our recursive $\mathsf{MerkleHash}(\cdot)$ implementation could be tricked into thinking an invalid Merkle path validates.

Algorithm 4 Creates and verifies append-only proofs

1: **function** $\mathsf{ProveAppendOnly}(\mathsf{pp}, \mathcal{S}_i, \mathcal{S}_j) \to \pi$
2: **if** $\mathsf{roots}(F_i) \subset \mathsf{roots}(F_j)$ **then return** $\perp$
3: Let $R = \{\text{roots} \in F_i \text{ but } \notin F_j\}$ and $r' \in \mathsf{roots}(F_j)$ be their ancestor root
4: $\pi \leftarrow \mathsf{ProvePath}(\mathcal{S}_j, r, r', \pi), \forall r \in R \quad \pi \leftarrow \mathsf{ProveRootAccs}(\mathcal{S}_i, \pi)$
5: **return** π
6: **function** $\mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$
7: **assert** $d_i(r) \neq \perp \Leftrightarrow r \in \mathsf{roots}(F_i)$ ▷ Is valid version i digest?
8: **assert** $d_j(r) \neq \perp \Leftrightarrow r \in \mathsf{roots}(F_j)$ ▷ Is valid version j digest?
9: **assert** $\forall r \in \mathsf{roots}(F_i) \cap \mathsf{roots}(F_j), d_i(r) = d_j(r)$
10: Let $R = \{\text{roots} \in F_i \text{ but } \notin F_j\}$ ▷ i.e., old roots with paths to new root
11: **for all** $r \in \mathsf{roots}(F_i)$ **do** ▷ Check proof gives correct old root accumulators
12: $(\cdot, a_r, \cdot, \cdot) \leftarrow \pi(r) \quad (h_{r|b}, \cdot, \cdot, \cdot) \leftarrow \pi(r|b), \forall b \in \{0, 1\}$
13: **assert** $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$
14: $\forall r \in R$, fetch h_r from $d_i(r)$ and update $\pi_{i,j}(r)$ with it
15: **assert** $\pi_{i,j}$ is well-formed Merkle proof for all roots in R
16: **assert** $\forall r \in R, \mathsf{VerPath}(d_j, r, \pi_{i,j})$

If k is stored at leaf ℓ in the AAS, $\mathsf{VerMemb}(\cdot)$ reconstructs ℓ's accumulator from k. Then, it checks if there's a valid Merkle path from ℓ to some root, verifying subset witnesses along the path via $\mathsf{VerPath}(\cdot)$ (see Algorithm 5). If k is not in the AAS, $\mathsf{VerMemb}(\cdot)$ verifies frontier proofs for k in each FCT in the forest via $\mathsf{VerFrontier}(\cdot)$ (see Algorithm 6).

Algorithm 5 Verifies a (non)membership proof

function $\mathsf{VerMemb}(\mathsf{VK}, d_i, k, b, \pi_k) \to \{T, F\}$
  Parse π_k as $\ell, \pi, (\chi_r)_{r \in \mathsf{roots}(F_i)}, (y_r, z_r)_{r \in \mathsf{roots}(F_i)}$
  if $b = 1$ **then** ▷ This is a membership proof being verified
    $(\cdot, a_\ell, \hat{a}_\ell) \leftarrow \mathsf{Accum}(\mathsf{P}(\{k\}))$ $h_\ell \leftarrow \mathcal{H}(\ell|\perp|a_\ell|\perp)$
    Update $\pi(\ell)$ with h_ℓ and accumulators a_ℓ and $\hat{a}_\ell$
    assert π is well-formed Merkle proof for leaf $\ell \wedge \mathsf{VerPath}(d_i, \ell, \pi)$
  else ▷ This is a non-membership proof being verified
    for all $r \in \mathsf{roots}(F_i)$ **do** ▷ Check FCTs
      $(\cdot, a_r, \cdot, \cdot) \leftarrow \pi(r)$ $(o_r, \cdot) \leftarrow \chi_r(\varepsilon)$
      $(h_{r|b}, \cdot, \cdot, \cdot) \leftarrow \pi(r|b), \forall b \in \{0, 1\}$
      assert $d_i(r) = \mathcal{H}(r|h_{r|0}|a_r|h_{r|1})$
      assert $e(a_r, y_r)e(o_r, z_r) = e(g, g) \wedge \mathsf{VerFrontier}(k, \chi_r)$
function $\mathsf{VerPath}(d_k, w, \pi) \to \{T, F\}$ ▷ Checks Merkle path from w to some root in d_k
  Let $r \in \mathsf{roots}(F_k)$ denote the ancestor root of w
  ▷ Walk path invariant: u is *not* a root node (but $\mathsf{parent}(u)$ might be)
  for $u \leftarrow w; u \neq r; u \leftarrow \mathsf{parent}(u)$ **do**
    $p \leftarrow \mathsf{parent}(u)$ ▷ Check subset witness and extractability (below)
    $(\cdot, a_u, \hat{a}_u, \pi_u) \leftarrow \pi(u)$ $(\cdot, a_p, \hat{a}_p, \cdot) \leftarrow \pi(p)$
    assert $e(a_u, \pi_u) = e(a_p, g) \wedge e(a_u, g^\tau) = e(\hat{a}_u, g)$
  assert $d_k(r) = \mathsf{MerkleHash}(\pi, r)$ ▷ Invariant: u equals r now
  assert $e(a_r, g^\tau) = e(\hat{a}_r, g)$ ▷ Is root accumulator extractable?
function $\mathsf{MerkleHash}(\pi, w) \to h_w$
  $(h_w, a_w, \cdot, \cdot) \leftarrow \pi(w)$
  if $h_w = \perp$ **then**
    return $\mathcal{H}(w|\mathsf{MerkleHash}(\pi, w|0)|a_w|\mathsf{MerkleHash}(\pi, w|1))$
  else
    return h_w

Frontier Algorithms. $\mathsf{CreateFrontier}(\cdot)$ creates a FCT level by level, starting from the leaves, given a set of frontier prefixes F. Given a key $k \notin \mathcal{S}_i$ and a root r, $\mathsf{ProveFrontier}(\cdot)$ returns a frontier proof for k in the FCT at root r. $\mathsf{VerFrontier}(\cdot)$ verifies a frontier proof for one of k's prefixes against a specific root FCT accumulator.

Algorithm 6 Manages FCT of a set

function CreateFrontier(F) $\to (\phi, \sigma)$
  $i \leftarrow 0$ $S_w \leftarrow \varnothing, \forall w$
  for $\rho \in F$ **do** ▷ First, build FCT leaves, with $g^{s-\mathcal{H}_{\mathbb{F}}(\rho)}$ for each prefix ρ
    $w \leftarrow \mathsf{bin}^{|F|}(i)$ $S_w \leftarrow \rho$ $i \leftarrow i+1$
    $(\phi_w, o, \hat{o}) \leftarrow \mathsf{Accum}(S_w)$ $\sigma(w) \leftarrow (o, \hat{o})$
  for $i \leftarrow \lceil \log |F| \rceil; i \neq 0; i \leftarrow i-1$ **do** ▷ Then, build FCT level by level
    $j \leftarrow 0$ $\mathsf{levelSize} \leftarrow 2^i$ $u \leftarrow \mathsf{bin}^{\mathsf{levelSize}}(0)$
    while $S_u \neq \varnothing$ **do** ▷ Merge sibling accumulators on level i
      $p \leftarrow \mathsf{parent}(u)$ $S_p \leftarrow S_u \cup S_{\mathsf{sibling}(u)}$ $j \leftarrow j+2$
      $(\phi_p, o, \hat{o}) \leftarrow \mathsf{Accum}(S_p)$ $\sigma(p) \leftarrow (o, \hat{o})$ $u \leftarrow \mathsf{bin}^{\mathsf{levelSize}}(j)$
  return $(\phi_\varepsilon, \sigma)$
function ProveFrontier($\mathcal{S}_i, r, k$) $\to \chi$
  Let ρ be the smallest prefix of k that is not in $\mathsf{P}(\mathbf{S}_r)$
  Let ℓ denote the leaf where $\boldsymbol{\sigma}_r(\ell) = g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))}$
  $\chi(\varepsilon) \leftarrow \boldsymbol{\sigma}_r(\varepsilon)$ ▷ Copy root FCT accumulator
  for $w \in \mathsf{path}[\ell, \varepsilon)$ **do** ▷ Copy path to ρ's FCT leaf
    $\chi(w) \leftarrow \boldsymbol{\sigma}_r(w)$
    if $\boldsymbol{\sigma}_r(\mathsf{sibling}(w)) \neq \perp$ **then**
      $\chi(\mathsf{sibling}(w)) \leftarrow \boldsymbol{\sigma}_r(\mathsf{sibling}(w))$
    else
      $\chi(\mathsf{sibling}(w)) \leftarrow (g, g^\tau)$
  return χ
function VerFrontier(k, χ) $\to \{T, F\}$
  ▷ Find leaf ℓ in χ with a prefix ρ for k, or fail.
  assert $\exists \ell, \rho$ such that $\rho \in \mathsf{P}(\{k\}) \wedge g^{(s-\mathcal{H}_{\mathbb{F}}(\rho))} = \chi(\ell)$
  assert $e(o, g^\tau) = e(\hat{o}_w, g)$ where $(o, \hat{o}) \leftarrow \chi(\varepsilon)$
  for $w \in \mathsf{path}[\ell, \varepsilon)$ **do** ▷ Verify ρ's membership in the FCT
    $(c_w, \hat{c}_w) \leftarrow \chi(w)$ $(s_w, \cdot) \leftarrow \chi(\mathsf{sibling}(w))$
    $(p_w, \cdot) \leftarrow \chi(\mathsf{parent}(w))$
    assert $e(c_w, s_w) = e(p_w, g) \wedge e(c_w, g^\tau) = e(\hat{c}_w, g)$

5.3.6 Security Proofs

Membership and append-only correctness follow from close inspection of the algorithms. Here, we prove our AAS construction offers membership and append-only security, as well as fork consistency.

5.3.6.1 Membership Security

Assume there exists a polynomial-time adversary $\mathcal{A}$ that produces digest d, element k and inconsistent proofs π, π' such that $\mathsf{VerMemb}(VK, d, k, 1, \pi)$ and $\mathsf{VerMemb}(VK, d, k, 0, \pi')$ both accept. We describe how $\mathcal{A}$ can either find a collision in $\mathcal{H}$ (used to hash the PCTs) or break

the q-SBDH assumption.

First, let us focus on the membership proof π, which consists of a path to k's leaf in some PCT of size 2^ℓ leaves. Let $a_0, a_1, \ldots, a_\ell$ be the prefix accumulators along this path (part of π), where a_0 is the leaf accumulator for element k with characteristic polynomial $A_0(x) = \prod_{c \in \mathsf{P}(k)}(x - \mathcal{H}_{\mathbb{F}}(c))$. Let $\pi_0, \ldots, \pi_{\ell-1}$ denote the corresponding subset witnesses, such that $e(a_j, g) = e(a_{j-1}, \pi_{j-1}), \forall j \in [\ell]$.

Second, let us consider the other (contradictory) non-membership proof π', which consists of a path to a FCT leaf storing a prefix ρ of k. Let $o_0, o_1, \ldots, o_{\ell'}$ be the frontier accumulators along this path, where o_0 is the leaf accumulator for ρ with characteristic polynomial $O_0(x) = x - \mathcal{H}_{\mathbb{F}}(\rho)$. Note that this FCT is of size $2^{\ell'}$, which might differ from 2^ℓ, the size of π's PCT. Let a^*_ℓ be the root prefix accumulator for this FCT's corresponding PCT, as contained in π' (see Algorithm 3).

When verifying π and π', both a_ℓ and a^*_ℓ are hashed (together with the two claimed hash values of their children) and the result is checked against the hash from digest d. Since verification of π and π' succeeds, if $a_\ell \neq a^*_\ell$ this would produce a collision in $\mathcal{H}$.

Else, we argue as follows. Each prefix accumulator $a_1, \ldots, a_\ell$ is accompanied by an extractability term $\hat{a}_1, \ldots, \hat{a}_\ell$, which the client checks as $e(a_j, g^\tau) = e(\hat{a}_j, g)$ for $j \in [\ell]$ (see Line 19 in Algorithm 5). Hence, from the q-PKE assumption, it follows that there exists a polynomial-time algorithm that, upon receiving the same input as $\mathcal{A}$, outputs polynomials $(A_j(x))_{j \in [\ell]}$ (in coefficient form) such that $g^{A_j(s)} = a_j$ with all but negligible probability. The same holds for all frontier accumulators $o_1, \ldots, o_{\ell'}$ and terms $\hat{o}_1, \ldots, \hat{o}_{\ell'}$ included in π', and let $(O_j(x))_{j \in [\ell']}$ denote their polynomials.

We distinguish two cases and analyze them separately:

(a) $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \nmid A_\ell(x)$ or $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \nmid O_{\ell'}(x)$
(b) $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid A_\ell(x)$ and $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid O_{\ell'}(x)$

For case (a), without loss of generality we will focus on the $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \nmid A_\ell(x)$ sub-case. (The proof for the second sub-case proceeds identically.) Observe that, by construction, $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid A_0(x)$ and, by assumption, $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \nmid A_\ell(x)$. Thus, there must exist some index $0 < i \leq \ell$ such that $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid A_{i-1}(x)$ and $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \nmid A_i(x)$. Note that i can be easily deduced given all $(A_j(x))_{j \in [\ell]}$. Therefore, by polynomial division there exist efficiently computable polynomials $q_i(x), q_{i-1}(x)$ and $\kappa \in \mathbb{F}_p$ such that: $A_{i-1}(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_{i-1}(x)$ and $A_i(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_i(x) + \kappa$.

Now, during the verification of the ith subset witness, it holds that:

$$
\begin{aligned}
e(a_i, g) &= e(a_{i-1}, \pi_{i-1}) \Leftrightarrow \\
e(g^{A_i(s)}, g) &= e(g^{A_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow \\
e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_i(s) + \kappa}, g) &= e(g^{(s-\mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow \\
e(g^{q_i(s) + \frac{\kappa}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) &= e(g^{q_{i-1}(s)}, \pi_{i-1}) \Leftrightarrow \\
e(g^{\frac{\kappa}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) &= e(g^{q_{i-1}(s)}, \pi_{i-1}) \cdot e(g^{-q_i(s)}, g) \Leftrightarrow \\
e(g^{\frac{1}{(s-\mathcal{H}_{\mathbb{F}}(\rho))}}, g) &= \left[e(g^{q_{i-1}(s)}, \pi_{i-1}) \cdot e(g^{-q_i(s)}, g) \right]^{\kappa^{-1}} .
\end{aligned}
$$

Hence, the pair $(\mathcal{H}_{\mathbb{F}}(\rho), \left[e(g^{q_{i-1}(s)}, \pi_{i-1}) \cdot e(g^{-q_i(s)}, g)\right]^{\kappa^{-1}})$ can be used to break the q-SBDH assumption.

In case (b), by assumption, $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid A_\ell(x)$ and $(x - \mathcal{H}_{\mathbb{F}}(\rho)) \mid O_{\ell'}(x)$. Therefore, by polynomial division there exist efficiently computable polynomials $q_A(x), q_o(x)$ such that: $A_\ell(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_A(x)$ and $O_{\ell'}(x) = (x - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_o(x)$. Let $\psi = (y, z)$ be the disjointness witness from π'. Since ψ verifies against accumulators a_ℓ and $o_{\ell'}$, it holds that:

$$\begin{aligned} e(a_\ell, y) \cdot e(o_{\ell'}, z) &= e(g, g) \Leftrightarrow \\ e(g^{A_\ell(s)}, y) \cdot e(g^{O_{\ell'}(s)}, z) &= e(g, g) \Leftrightarrow \\ e(g^{(s - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_A(s)}, y) \cdot e(g^{(s - \mathcal{H}_{\mathbb{F}}(\rho)) \cdot q_o(s)}, z) &= e(g, g) \Leftrightarrow \\ e(g^{q_A(s)}, y) \cdot e(g^{q_o(s)}, z) &= e(g, g)^{\frac{1}{(s - \mathcal{H}_{\mathbb{F}}(\rho))}}. \end{aligned}$$

Thus, the pair $(\mathcal{H}_{\mathbb{F}}(\rho), e(g^{q_A(s)}, y) \cdot e(g^{q_o(s)}, z))$ can again be used to break the q-SBDH assumption.

5.3.6.2 Append-only Security

We can prove append-only security with the same techniques used above. Let ρ be the prefix of k used to prove non-membership w.r.t. the new digest d_j. The membership proof for k w.r.t. the old digest d_i again involves a series of prefix accumulators whose corresponding polynomials can be extracted. By our previous analysis, $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ must divide the polynomial extracted for the root prefix accumulator in d_i, otherwise the q-SBDH assumption can be broken. Continuing on this sequence of subset witnesses, the append-only proof "connects" this old root prefix accumulator to a new root prefix accumulator in d_j. By the same argument, $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ must also divide the polynomial extracted for this new root prefix accumulator from d_j. Since non-membership also verifies, $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ must divide the extracted polynomial for the root frontier accumulator in d_j, or else q-SBDH can be broken. Finally, we apply the same argument as case (b) above, since $(x - \mathcal{H}_{\mathbb{F}}(\rho))$ divides both these polynomials and we have a disjointness witness for their accumulators, again breaking q-SBDH.

5.3.6.3 Fork Consistency

Assume there exists a polynomial-time adversary $\mathcal{A}$ that breaks fork consistency, producing digests $d_i \neq d_i'$ with append-only proofs π_i, π_i' to a new digest d_j. Since $d_i \neq d_i'$, there exists a root r such that its hash h_r in d_i differs from its hash h_r' in d_i'. Since d_i and d_i' get "joined" into d_j, let $r^* \neq r$ denote the ancestor root of r in d_j. (Note that $r^* \neq r$, since VerAppendOnly always makes sure that common roots between an old digest and a new digest have the same hash.) Now, note that both proofs π_i, π_i' are Merkle proofs from node r to r^*. Importantly, because every node w is hashed together with its label w (as $h_w = \mathcal{H}(w, h_{w|0}, a_w, h_{w|1})$), the two Merkle proofs take the same path (i.e., $\mathsf{path}[r, r^*]$)! In other words, the adversary produced two Merkle proofs that (1) verify against the same digest d_j, (2) take the same path to the same leaf r, but (3) attest for different leaf hashes h_r and h_r'. This breaks Merkle hash tree security and can be used to produce a collision in $\mathcal{H}$.

5.4 AAS from RSA Accumulators

In this section, we change our bilinear-based AAS from Section 5.3, which we dub $\mathsf{AAS}_{\mathsf{bilinear}}$, into an RSA-based AAS, which we dub $\mathsf{AAS}_{\mathsf{rsa}}$. This transformation is natural, since our $\mathsf{AAS}_{\mathsf{bilinear}}$ design is not dependent on the implementation of the underlying accumulator. In fact, any accumulator that supports subset witnesses and disjointness witnesses suffices to implement our design. Even better, RSA accumulators' $\Theta(n \log n)$ time to precompute all membership witnesses will reduce the asymptotic frontier proof sizes in our AAS from logarithmic to constant.

Before introducing $\mathsf{AAS}_{\mathsf{rsa}}$, we first describe how to enhance RSA accumulators with subset witnesses and disjointness witnesses. Our enhancements build upon Boneh et al.'s [31, 32] PoKE protocols (see Section 4.2). We do not prove security of these new RSA accumulator witnesses. Instead, we prove our AAS construction based on these witnesses, is secure in Section 5.4.4.

5.4.1 Subset and Disjointness Witnesses for RSA Accumulators

Boneh et al. present a *union witness* for proving that two RSA accumulators over sets A, B have been "unioned" into a single RSA accumulator over $A \cup B$. The witness size is three group elements and two λ-bit numbers. This union witness can be re-purposed as a subset witness between two accumulators over sets $A \subseteq B$. Since this witness is rather large, we introduce a new, smaller-sized subset witness that only uses one PoKE2 proof (see Section 4.2.4).

Subset Witnesses. Given sets $A \subseteq B$, and their RSA accumulators $a = \mathsf{acc}(A), b = \mathsf{acc}(B)$, naively proving that $A \subseteq B$ would involve sending all elements in their difference $B \backslash A$. Then, the verifier would check this (potentially very large) *subset witness* as follows:

$$a^{\prod_{e \in B \backslash A} e} = b$$

We immediately notice that the PoKE2 protocols can save the verifier time and communication in verifying the above equation. Let $x = \prod_{e \in B \backslash A} e$ be the exponent from the equation above. Then, the prover can use the PoKE2 protocol to prove that it knows x such that $a^x = b$, without having to send x to the verifier and without the verifier having to perform the (potentially very expensive) a^x exponentiation. In other words, we get a *constant-sized* subset witness for $A \subseteq B$ that can be verified in constant-time. Finally, this witness can be computed in $\Theta(|B \backslash A|)$ time.

Disjointness Witnesses. Using similar ideas, we also construct a *disjointness witness* for RSA accumulators. Given sets $A \cap B = \varnothing$ with RSA accumulators a, b, we can compute Bézout coefficients x, y such that:

$$x \left(\prod_{e \in A} e \right) + y \left(\prod_{e \in B} e \right) = 1$$

Naively, a (potentially large) disjointness witness would consist of the Bézout coefficients x, y which are as large as $\prod_{e \in B} e$ and $\prod_{e \in A} e$, respectively. The verifier would check the equality from above holds "in the exponent":

$$a^x b^y = g \Leftrightarrow \tag{5.1}$$

$$\left(g^{\prod_{e \in A} e}\right)^x \left(g^{\prod_{e \in B} e}\right)^y = g \Leftrightarrow \tag{5.2}$$

$$g^{x \prod_{e \in A} e + y \prod_{e \in B} e} = g \Leftrightarrow \tag{5.3}$$

$$x \left(\prod_{e \in A} e\right) + y \left(\prod_{e \in B} e\right) = 1. \tag{5.4}$$

Fortunately, PoKE2 proofs can be used to avoid sending x, y, resulting in a constant-sized disjointness witness. Specifically, the prover sends two PoKE2 proofs, one for $a^x = u$ and another for $b^y = w$. The verifier checks the two PoKE2 proofs and then checks that $u \cdot w = g$ to make sure Equation (5.2) holds. To compute the witness, the prover spends $\Theta(n \log^2 n \log \log n)$ time to compute Bézout coefficients [188], where $n = \max(|A|, |B|)$, and $\Theta(n)$ time to compute the two PoKE2 proofs. Finally, our disjointness witness can be made smaller by aggregating the two PoKE2 proofs [32].

5.4.2 From Bilinear-based to RSA-based AAS

Replacing the bilinear accumulator in $\mathsf{AAS}_{\mathsf{bilinear}}$ (see Section 5.3) with an RSA accumulator (see Section 4.3.2) together with some adjustments gives us an RSA-based AAS which we call $\mathsf{AAS}_{\mathsf{rsa}}$. This construction has several advantages over $\mathsf{AAS}_{\mathsf{bilinear}}$. First, it only requires constant-sized public parameters. Second, it does not inherently require a trusted setup phase (see Section 4.4). Third, its (non)membership proof size decreases from $O(\log^2 n)$ to $O(\log n)$. Finally, at the level of implementation, our memory consumption should also decrease, since FCTs are no longer required.

At the same time, $\mathsf{AAS}_{\mathsf{rsa}}$ inherits some of the disadvantages of RSA accumulators (see Section 4.3.2). First, because group elements in hidden-order groups are larger than in prime (known) order groups, our subset witnesses will be slightly larger than in $\mathsf{AAS}_{\mathsf{bilinear}}$. Second, appends will be slower because RSA accumulator and witness computation is slower. Third, even though we showed how to extend RSA accumulators with subset and disjointness witnesses, our witnesses incur the overheads of PoKE2 proofs (see Section 4.2.4). Finally, RSA accumulators require elements to be primes, which involves expensive hashing.

In the rest of this section, we highlight the differences between $\mathsf{AAS}_{\mathsf{bilinear}}$ and $\mathsf{AAS}_{\mathsf{rsa}}$.

Hashing to Primes. A key difference in $\mathsf{AAS}_{\mathsf{rsa}}$ is that prefixes of appended keys need to be hashed to their *prime representatives* via a hash function $\mathcal{H}_{\mathsf{prime}} : \{0,1\}^* \to \mathsf{Primes}(2\lambda)$. This is because, when enhanced with disjointness witnesses, RSA accumulators require the accumulated elements to be primes (see Section 4.3.2). This is similar to $\mathsf{AAS}_{\mathsf{bilinear}}$, which also requires hashing the prefixes of keys, but to a finite field $\mathbb{F}_p$. Although, in $\mathsf{AAS}_{\mathsf{rsa}}$, the hashing of prefixes to primes is much more expensive since it involves primality testing. For example, hashing to a prime takes ≈ 1 millisecond while hashing to $\mathbb{F}_p$ takes ≤ 1 microsecond.

Constant-sized Public Parameters and (No) Trusted Setup. First, since RSA accumulators have constant-sized public parameters, $\mathsf{AAS}_{\mathsf{rsa}}$ has constant-sized public parameters for the server and clients. In contrast, $\mathsf{AAS}_{\mathsf{bilinear}}$ has q-SDH public parameters with $q = \Theta(\lambda n)$ for the server and $q = \Theta(\lambda)$ for the client. Second, if the RSA accumulator is instantiated over *class groups*, then the AAS scheme does not require a trusted setup (see Section 4.4). Even if instantiated over $\mathbb{Z}_N^*$, the trusted setup ceremony would be much simpler. This makes $\mathsf{AAS}_{\mathsf{rsa}}$ much easier to instantiate in practice.

Logarithmic Append-only and (Non)membership Proofs. First, recall that $\mathsf{AAS}_{\mathsf{bilinear}}$ used *Prefix Communion Trees* (PCTs) to precompute all *logarithmic-sized* membership & append-only proofs and used *Frontier Communion Trees* (FCTs) to precompute all non-membership proofs. $\mathsf{AAS}_{\mathsf{rsa}}$ does the same, except it optimizes FCTs and, optionally, PCTs. First, given the frontier $F(S)$ of a set S, we immediately notice that we can use RSA fast witness precomputation (see Section 4.3.2) to obtain all *constant-sized* frontier proofs in $\Theta(|F(S)| \log |F(S)|)$ time. Looked at differently, we obtain an FCT of height one: the root is the RSA frontier accumulator of S, while the leaves are the elements of S and their RSA membership witnesses. As a consequence, the AAS non-membership proof, which consists of $O(\log n)$ frontier proofs, is now $O(\log n)$-sized, an asymptotic improvement over $\mathsf{AAS}_{\mathsf{bilinear}}$. We believe this could also translate to a concrete improvement in proof sizes.

Second, the same technique can (optionally) be applied for the PCT, to precompute constant-sized RSA witnesses for all prefixes in $\mathsf{P}(S)$. Then, an AAS membership proof for k would consist of $\Theta(\lambda)$ witnesses, one for each prefix of k. These witnesses can be aggregated efficiently using techniques from [31] into a single constant-sized witness, reducing the AAS membership proof size to $O(1)$, which is an asymptotic improvement over $\mathsf{AAS}_{\mathsf{bilinear}}$. However, to save computation time, the server can choose not to deploy this improvement and use traditional PCTs only, maintaining logarithmic-sized AAS membership proofs. (Indeed, our security proof from Section 5.4.4 assumes traditional PCT-based membership proofs for $\mathsf{AAS}_{\mathsf{rsa}}$.) In contrast, for the frontier, we stress that $\mathsf{AAS}_{\mathsf{rsa}}$ always uses the RSA witness precomputation trick to reduce AAS non-membership proof to logarithmic size.

5.4.3 Asymptotic Analysis

As in Section 5.3.4, suppose we have a *worst-case* AAS with $n = 2^i - 1$ elements, but based on RSA accumulators.

Append-time. As before, consider the time $T(n)$ to create an AAS over a set S with $n = 2^\ell$ elements (without loss of generality). Then, $T(n)$ can be broken into:

- The time to create each children AAS of size $n/2$ (i.e., $2T(n/2)$).
- The time to merge these two children, which can be broken into:
  - Computing an RSA prefix accumulator for S in $\Theta(\lambda n)$ time.
  - Computing subset witnesses between S's prefix accumulator and each of its two children's prefix accumulators in $\Theta(\lambda n)$ time.

– Computing the FCT of S, which requires $\Theta(\lambda n \log^2 n \log\log n)$ time for the Bézout coefficients and $\Theta(\lambda n \log n)$ time for the frontier proofs (via the RSA fast membership witness precomputation trick from Section 4.3.2).

Thus, the *amortized* time for one append is $O(\lambda \log^3 n \log\log n)$, which is higher by a factor of $\log\log n$ than in the bilinear AAS (see Section 5.3.4). Appends can be de-amortized into *worst-case* time using generic techniques [165, 166].

Space, Digests, Append-only Proofs and Membership Proofs. Here, $\mathsf{AAS}_{\mathsf{rsa}}$ has the same overheads as $\mathsf{AAS}_{\mathsf{bilinear}}$, which are discussed in Section 5.3.4:

- $\Theta(\lambda n)$ space.
- $O(\log n)$ digests, which can be made $\Theta(1)$.
- $\Theta(\log n)$ membership and append-only proofs.

However, $\mathsf{AAS}_{\mathsf{rsa}}$ can support $\Theta(1)$ membership witnesses (see Section 5.4.2), but only with extra computation during appends and by keeping digests of size $O(\log n)$.

Non-membership Proof Size. Recall that $\mathsf{AAS}_{\mathsf{bilinear}}$ had $\Theta(\log^2 n)$ sized proofs because, in the worst-case, a frontier proof in each of the $\Theta(\log n)$ FCTs would need to be sent. Fortunately, since frontier proofs in our case are $\Theta(1)$-sized, this reduces our non-membership proof size to $\Theta(\log n)$.

Public Parameters. Unlike $\mathsf{AAS}_{\mathsf{bilinear}}$, $\mathsf{AAS}_{\mathsf{rsa}}$ has constant-sized public parameters, both on the server and client side.

5.4.4 Security Proofs

In these proofs, we ignore Merkle hashing, which is done for fork consistency purposes. Instead, we assume that the digest of the AAS only contains the root prefix accumulators in the forest (rather than Merkle root hashes). The proof with Merkle hashing would not differ much. For example, we give such a proof for the bilinear AAS in Section 5.3.6.1.

We also assume that the RSA witness precomputation trick is not used to precompute constant-sized AAS membership proofs. (This assumption actually makes the security proof slightly more complex.) However, recall that this trick is always used to precompute all constant-sized frontier proofs (and reduce AAS non-membership proof size to logarithmic).

5.4.4.1 Append-only Security

Suppose an adversary $\mathcal{A}$ breaks append-only security (see Definition 5.2.5) and outputs:

- Two digests $d \neq d'$,
- A valid membership proof π for a key k in d,
- A valid append-only proof $\pi^{\subseteq}$ from d to d',
- A valid non-membership proof π' for the same key k in d'.

Then, we can build another adversary $\mathcal{B}$ that breaks the Strong RSA assumption (see Section 4.1.1) as follows. First, $\mathcal{B}$ runs $\mathcal{A}$ and obtains $(d, d', \pi^{\subseteq}, k, \pi, \pi')$.

Since π proves that k is in the AAS with digest d, then π will consist of a path of accumulators with subset witnesses, from k's leaf to a root accumulator in d. Let a_ℓ denote this accumulator and $\ell + 1$ denote the length of the path (i.e., the number of nodes, including the leaf and the root node). Let a_0 be the leaf accumulator over all prefixes of k. Note that the verifier, in this case $\mathcal{B}$, reconstructs $a_0 = g^z, z = \prod_{\rho \in \mathsf{P}(k)} \mathcal{H}_{\mathsf{prime}}(\rho)$ from the key k.

Let $(a_i)_{i \in [1,\ell)}$ and $(\pi_i^{\in})_{i \in [0,\ell)}$ denote the accumulators and subset witnesses, respectively, from the membership proof π. Recall that a subset witness $\pi_i^{\in}$ is a PoKE2 proof that the prover knows an $x_i^{\in} \in \mathbb{Z}$ such that $a_i^{x_i^{\in}} = a_{i+1}$ (for all $i \in [0, \ell)$). In particular, since this is a proof of *knowledge* [108], an *extractor* exists that $\mathcal{B}$ can use to obtain the exponents $x_i^{\in} \in \mathbb{Z}, \forall i \in [0, \ell)$. Let $x^{\in} = \prod_{i \in [0,\ell)} x_i^{\in}$, which $\mathcal{B}$ can compute, and note that $(a_0)^{x^{\in}} = a_\ell$.

The append-only proof $\pi_{i,j}$ "connects" all root accumulators from digest d to one root accumulator in d' via subset witnesses. Let this root accumulator in d' be denoted by $a_{\ell'}$, where $\ell' \geq \ell$ is the height of the tallest tree in d'. (Importantly, it could be that $a_\ell = a'_\ell$; e.g., $\pi_{2,3}$ in Figure 5-4.) Then, amongst other paths, $\pi_{i,j}$ contains a path of accumulators $(a_i)_{i \in (\ell, \ell')}$ with subset witnesses $(\pi_i^{\subseteq})_{i \in [\ell, \ell')}$. As before, $\pi_i^{\subseteq}$ is a PoKE2 proof of knowledge for an $x_i^{\subseteq} \in \mathbb{Z}$ such that $a_i^{x_i^{\subseteq}} = a_{i+1}$ and this $x_i^{\subseteq}$ can be extracted by $\mathcal{B}$ (for all $i \in [\ell, \ell')$). Let $x^{\subseteq} = \prod_{i \in [\ell, \ell')} x_i^{\subseteq}$, which $\mathcal{B}$ can compute, and note that $(a_\ell)^{x^{\subseteq}} = a_{\ell'}$.

Since π' proves that k is *not* in the AAS with digest d', then π' has to prove that k is not in any of the root accumulators from d'. In particular, it has to prove non-membership of k in $a_{\ell'}$. For this, π' includes the frontier accumulator $f_{\ell'}$ corresponding to $a_{\ell'}$ together with a disjointness witness $(u, v, \mathsf{poke}_u, \mathsf{poke}_v)$, which proves knowledge of Bézout coefficients α, β such that $(a_{\ell'})^\alpha = u$ and $(f_{\ell'})^\beta = v$ where $u \cdot v = g$. As before, $\mathcal{B}$ can extract α and β. Finally, π' proves membership in f_ℓ of a prefix ρ of k with prime representative $e = \mathcal{H}_{\mathsf{prime}}(\rho)$. In other words, π' includes an RSA membership witness w such that $w^e = f_{\ell'}$.

At this point, recall that $\mathcal{B}$ computed $a_0 = g^z$ where $z = \prod_{\rho \in \mathsf{P}(k)} \mathcal{H}_{\mathsf{prime}}(\rho)$. Importantly, the prime representative e divides z (since it is one of the prefixes of k) and $\mathcal{B}$ can compute $c = z/e$. In other words, $a_0 = g^{ec}$.

Now, $\mathcal{B}$ can break Strong RSA by noticing that:

$$
\begin{aligned}
(a_{\ell'})^\alpha (f_{\ell'})^\beta &= g \Leftrightarrow \\
\left((a_\ell)^{x^{\subseteq}}\right)^\alpha (w^e)^\beta &= g \Leftrightarrow \\
\left((a_0)^{x^{\in}}\right)^{\alpha x^{\subseteq}} w^{e\beta} &= g \Leftrightarrow \\
(a_0)^{\alpha x^{\in} x^{\subseteq}} w^{e\beta} &= g \Leftrightarrow \\
(g^{ec})^{\alpha x^{\in} x^{\subseteq}} w^{e\beta} &= g \Leftrightarrow \\
g^{ec\alpha x^{\in} x^{\subseteq}} w^{e\beta} &= g \Leftrightarrow \\
\left(g^{c\alpha x^{\in} x^{\subseteq}} w^\beta\right)^e &= g.
\end{aligned}
$$

Thus, $\mathcal{B}$ computes $y = g^{c\alpha x^{\in} x^{\subseteq}} w^\beta$. Since e is a prime and $y^e = g$, $\mathcal{B}$ breaks Strong RSA (see Definition 4.1.1).

5.4.4.2 Membership Security

This proof proceeds in the same style as the append-only security proof from Section 5.4.4.1. Suppose an adversary $\mathcal{A}$ breaks membership security (see Definition 5.2.3) and outputs:

- A digests d,
- A valid membership proof π for a key k in d,
- A valid non-membership proof π' for the same key k in d.

Then, we can build another adversary $\mathcal{B}$ that breaks the Strong RSA assumption (see Section 4.1.1) as follows. First, $\mathcal{B}$ runs $\mathcal{A}$ and obtains (d, k, π, π').

Our reasoning is the same as in Section 5.4.4.1. Let a_0 denote the leaf accumulator over k's prefixes and a_ℓ denote the root accumulator where k's membership is proven. Since π proves that k is in the AAS with digest d, ultimately, the adversary $\mathcal{B}$ can extract and compute $x^\in \in \mathbb{Z}$ such that $(a_0)^{x^\in} = a_\ell$. Since π' proves that k is *not* in the AAS with digest d, then π' has to prove non-membership of k in a_ℓ. To do this, π' proves membership of a prefix ρ of k with prime representative e in the frontier accumulator f_ℓ which is provably disjoint from a_ℓ. Specifically, π' contains w such that $w^e = f_\ell$. As in Section 5.4.4.1, from the disjointness witness between a_ℓ and f_ℓ, $\mathcal{B}$ extracts Bézout coefficients α, β such that $(a_\ell)^\alpha = u$ and $(f_\ell)^\beta = v$ where $u \cdot v = g$. Recall from Section 5.4.4.1 that $\mathcal{B}$ computes c and $a_0 = g^{ec}$ where $c = \left(\prod_{\rho \in \mathsf{P}(k)} \mathcal{H}_{\mathsf{prime}}(\rho)\right)/e$.

Now, $\mathcal{B}$ can break Strong RSA by noticing that:

$$
\begin{aligned}
(a_\ell)^\alpha \, (f_\ell)^\beta &= g \Leftrightarrow \\
\left((a_0)^{x^\in}\right)^\alpha w^{e\beta} &= g \Leftrightarrow \\
(a_0)^{\alpha x^\in} \, w^{e\beta} &= g \Leftrightarrow \\
(g^{ec})^{\alpha x^\in} \, w^{e\beta} &= g \Leftrightarrow \\
g^{ec\alpha x^\in} w^{e\beta} &= g \Leftrightarrow \\
\left(g^{c\alpha x^\in} w^\beta\right)^e &= g.
\end{aligned}
$$

Thus, $\mathcal{B}$ computes $y = g^{c\alpha x^\in} w^\beta$. Since e is a prime and $y^e = g$, $\mathcal{B}$ breaks Strong RSA (see Definition 4.1.1).

Chapter 6

Append-only Authenticated Dictionaries (AAD)

6.1 Overview

In this section, we generalize the notion of an append-only authenticated set from Chapter 5 to an *append-only authenticated dictionary* (AAD). Recall that an AAS stores *elements* and supports (non)membership queries of the form "Is $e \in S$?" In contrast, an AAD stores *key-value pairs* and supports *lookups* of the form "Is V the complete set of values for key k?" In other words, an AAD maps a *key* k to a multiset of *values* V in an append-only fashion. Specifically, once a value has been added to a key, it cannot be removed nor changed. For example, if a key is a domain name and its values are certificates for that domain, then an AAD can be used as a Certificate Transparency (CT) log. In general, keys and values can have any application-specific type.

Naturally, we need new security notions that capture the additional functionality of dictionaries over sets. Specifically, in an AAS, no malicious server can simultaneously produce accepting proofs of membership and non-membership for the same element e with respect to the same digest. In contrast, in an AAD, no malicious server can simultaneously produce accepting proofs for two different sets of values V, V' for a key k with respect to the same digest. This captures the related notion for an AAS since one of the sets of values may be empty (indicating k has never been registered in the dictionary) and the other non-empty. Similarly, the security definitions for the "append-only" property can also be adjusted.

6.2 Definitions

6.2.1 Server-side API

The untrusted server implements:

$\mathsf{Setup}(1^\lambda, B) \to \mathsf{pp}, \mathsf{VK}$. Randomized algorithm that returns public parameters pp used by the server and a *verification key* VK used by clients. Here, λ is a security parameter and B is an upper-bound on the number of elements n in the dictionary (i.e., $n \leq B$).

$\mathsf{Append}(\mathsf{pp}, \mathcal{D}_i, d_i, k, v) \to \mathcal{D}_{i+1}, d_{i+1}$. Deterministic algorithm that appends a new key-value pair (k, v) to the version i dictionary, creating a new version $i+1$ dictionary. Succeeds only if the dictionary is not full (i.e., $i+1 \leq B$). Returns the new authenticated dictionary $\mathcal{D}_{i+1}$ and its digest d_{i+1}.

$\mathsf{ProveLookup}(\mathsf{pp}, \mathcal{D}_i, k) \to V, \pi_{k,V}$. Deterministic algorithm that generates a proof $\pi_{k,V}$ that V is the *complete* multiset of values for key k. In particular, when $\mathcal{D}_i(k) = \varnothing$, this is a proof that key k has no values. Finally, the server cannot construct a fake proof $\pi_{k,V'}$ for the wrong V', including for $V' = \varnothing$.

$\mathsf{ProveAppendOnly}(\mathsf{pp}, \mathcal{D}_i, \mathcal{D}_j) \to \pi_{i,j}$. Deterministic algorithm that proves $\mathcal{D}_i$ is a subset of $\mathcal{D}_j$. Generates an *append-only proof* $\pi_{i,j}$ that all key-value pairs in $\mathcal{D}_i$ are also present and unchanged in $\mathcal{D}_j$. Importantly, a malicious server who removed or changed keys from $\mathcal{D}_j$ that were present in $\mathcal{D}_i$ cannot construct a valid append-only proof.

6.2.2 Client-side API

Clients implement:

$\mathsf{VerLookup}(\mathsf{VK}, d_i, k, V, \pi) \to \{T, F\}$. Deterministic algorithm that verifies proofs returned by $\mathsf{ProveLookup}(\cdot)$ against the digest d_i at version i of the dictionary. When $V \neq \varnothing$, verifies that V is the complete multiset of values for key k, ensuring no values have been left out and no extra values were added. When $V = \varnothing$, verifies that key k is not mapped to any value.

$\mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_{i,j}) \to \{T, F\}$. Deterministic algorithm that ensures a dictionary remains append-only. Verifies that $\pi_{i,j}$ correctly proves that the dictionary with digest d_i is a subset of the dictionary with digest d_j. Also, verifies that d_i and d_j are digests of dictionaries at version i and j, respectively.

6.2.3 Correctness and Security

Consider an ordered sequence of n key-value pairs $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$. Note that the same key (or key-value pair) can occur multiple times in the sequence. Let $\mathcal{D}', d' \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{D}, d, (k_i, v_i)_{i\in[n]})$ denote a sequence of $\mathsf{Append}(\cdot)$ calls arbitrarily interleaved with other $\mathsf{ProveLookup}(\cdot)$ and $\mathsf{ProveAppendOnly}(\cdot)$ calls such that $\mathcal{D}', d' \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{D}_{n-1}, d_{n-1}, k_n, v_n)$, $\mathcal{D}_{n-1}, d_{n-1} \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{D}_{n-2}, d_{n-2}, k_{n-1}, v_{n-1})$, $\ldots$, $\mathcal{D}_1, d_1 \leftarrow \mathsf{Append}(\mathsf{pp}, \mathcal{D}, d, k_1, v_1)$. Let D_n denote the *corresponding dictionary* obtained after appending each $(k_i, v_i)_{i\in[n]}$ in order. Finally, let $\mathcal{D}_0$ denote an empty authenticated dictionary with (empty) digest d_0.

Definition 6.2.1 (Append-only Authenticated Dictionary)**.** (Setup, Append, $\mathsf{ProveLookup}$, $\mathsf{ProveAppendOnly}$, $\mathsf{VerLookup}$, $\mathsf{VerAppendOnly}$) is a secure append-only authenticated dictionary (AAD) if $\forall$ upper-bounds $B = \mathsf{poly}(\lambda)$ it satisfies the following properties:

Definition 6.2.2 (AAD Lookup Correctness)**.** $\forall n \leq B$, $\forall$ sequences $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$

with corresponding dictionary D_n, $\forall$ keys $k \in \mathcal{K}$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B),\\ (\mathcal{D}, d) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{D}_0, d_0, (k_i, v_i)_{i\in[n]}),\\ (V, \pi) \leftarrow \mathsf{ProveLookup}(\mathsf{pp}, \mathcal{D}, k) :\\ V = D_n(k) \wedge \mathsf{VerLookup}(\mathsf{VK}, d, k, V, \pi) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Observation: Note that this definition compares the returned multiset V with the "ground truth" in D_n and thus provides lookup correctness. Also, it handles non-membership correctness since V can be the empty set. Finally, the definition handles all possible orders of inserting key-value pairs.

Definition 6.2.3 (AAD Lookup Security)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B),\\ (d, k, V \neq V', \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{VK}) :\\ \mathsf{VerLookup}(\mathsf{VK}, d, k, V, \pi) = T \wedge\\ \mathsf{VerLookup}(\mathsf{VK}, d, k, V', \pi') = T\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This definition captures the lack of any "ground truth" about what was inserted in the dictionary, since there is no trusted source in our model. Nonetheless, given a fixed digest d, our definition prevents *all* equivocation attacks about the complete multiset of values of a key, including the special case where the server equivocates about the key being present (i.e., $V \neq \varnothing$ and $V' = \varnothing$).

Definition 6.2.4 (AAD Append-only Correctness)**.** $\forall n \leq B$, $\forall$ sequences $(k_i \in \mathcal{K}, v_i \in \mathcal{V})_{i\in[n]}$ where $n \geq 2$

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B)\\ (\mathcal{D}_m, d_m) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{D}_0, d_0, (k_i, v_i)_{i\in[m]}),\\ (\mathcal{D}_n, d_n) \leftarrow \mathsf{Append}^+(\mathsf{pp}, \mathcal{D}_m, d_m, (k_j, v_j)_{j\in[m+1,n]}),\\ \pi \leftarrow \mathsf{ProveAppendOnly}(\mathsf{pp}, \mathcal{D}_m, \mathcal{D}_n) :\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_m, m, d_n, n, \pi) = T\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 6.2.5 (AAD Append-only Security)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$,

$$\Pr\left[\begin{array}{c}(\mathsf{pp},\mathsf{VK}) \leftarrow \mathsf{Setup}(1^\lambda, B)\\ (d_i, d_j, i < j, \pi_a, k, V, V', \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}, \mathsf{VK}) :\\ \mathsf{VerAppendOnly}(\mathsf{VK}, d_i, i, d_j, j, \pi_a) = T \wedge\\ \mathsf{VerLookup}(\mathsf{VK}, d_i, k, V, \pi) = T \wedge\\ \mathsf{VerLookup}(\mathsf{VK}, d_j, k, V', \pi') = T \wedge\\ V \not\subset V' \wedge V \neq V'\end{array}\right] \leq \mathsf{negl}(\lambda)$$

Observation: This definition ensures that values can only be added to a key and can never be removed nor changed.

Definition 6.2.6 (AAD Fork Consistency)**.** This definition stays the same as for an AAS (see Definition 5.2.6).

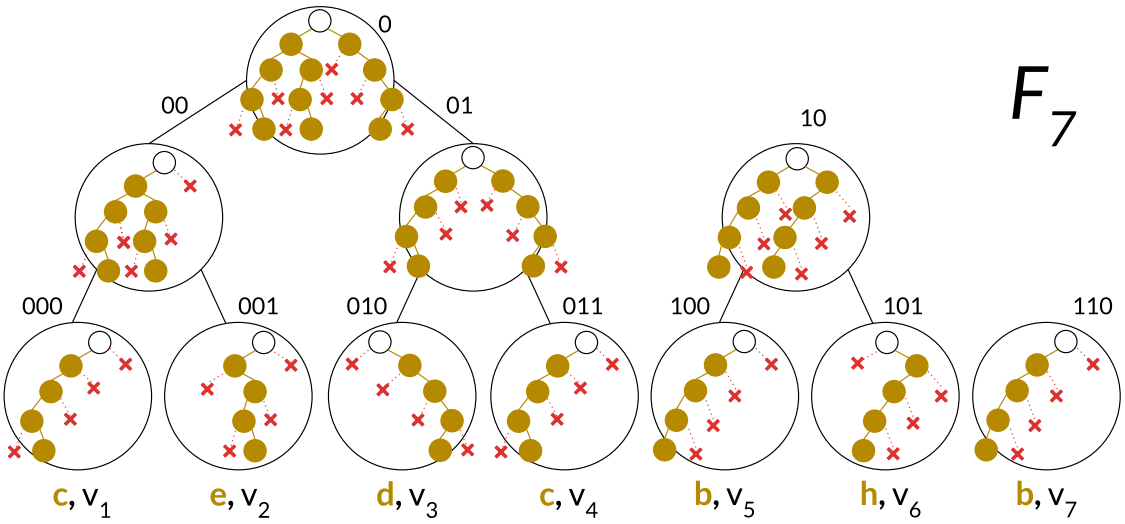


Figure 6-1: A dynamic AAD for dictionary $\{(b, v_5), (b, v_7), (c, v_1), (c, v_4), (d, v_3), (e, v_2), (h, v_6)\}$ with $\lambda = 2$. Unlike the AAS from Figure 5-5, this AAD stores key-value pairs in the forest leaves. Furthermore, *every node* in the forest now stores a Frontier Communion Tree (FCT). Similar to the AAS, all accumulators are built over the prefixes of the keys. This way, the accumulators can be used to "provably-guide" a search for all the values of a key.

6.3 AAD from (Any) Accumulator

In this section, we give two approaches of building AADs from any cryptographic accumulator (see Section 4.3). In the first approach, we modify our AAS construction from Section 5.3 to keep track of key-value pairs in the forest leaves, rather than just keys. Unfortunately, this approach has the disadvantage of requiring $O(\lambda n \log n)$ space. In our second approach, we reduce the space to $O(\lambda n)$ at the cost of making appends slight slower and doubling the size of the public parameters. Append-only proof sizes remain the same in both constructions. Since we only implement this second approach (see Section 6.4), we do not investigate the concrete differences in lookup proof sizes.

6.3.1 From AAS to AAD

We can easily modify our AAS approach from Section 5.3 to obtain an AAD as depicted in Figure 6-1. First, the leaves in the forest will now store *key-value pairs*, since we want dictionaries rather than sets. As before, each node in the forest will store a prefix accumulator, but the accumulator will be computed only over the keys in its subtrees, even though the leaves store key-value pairs. Thus, each tree in the forest can be regarded as a PCT, which stores key-value pairs as leaves but only accumulates the keys. This will be very useful for proving lookups.

Second, all nodes in the forest, not just root nodes, need to store an FCT. (In contrast, in our AAS, only root nodes stored FCTs, and internal nodes discarded them after merges.) The FCTs enable the server to easily prove that a key k is *not* in a node's subtree. This is necessary for proving lookups. On the other hand, FCTs at every node increase the space on

the server to $O(\lambda n \log n)$. Finally, append-only proofs remain the same as in the AAS. We can now detail exactly how lookup proofs work.

6.3.1.1 Proving Lookups

Consider lookups for a key k with a single value v. Equipped with the ability of proving non-membership of k in any node in the forest, the server can now easily prove such lookups. First, the server shows (k, v) is in a PCT using a proof path. Second, to show that k has no other values in that tree, the server proves k is *not* in any of the subtrees rooted at the sibling nodes along the proof path. We call these subtrees *missing subtrees*, since only their roots are included in the PCT proof path (as sibling nodes). For every missing subtree, the server gives a frontier proof for k not being there. Finally, for all other PCTs in the forest, the server gives a frontier proof that k is not in that PCT.

We can now generalize. Suppose k has (zero or more) values V "spread out" across t of the $O(\log n)$ trees in the forest. The lookup proof consists of:

- For each value $v \in V$, a PCT proof path to (k, v) in the forest. These paths add $O(|V| \log n)$ overhead and form what we call a *pruned forest*.
- For each missing subtree in this pruned forest, a frontier proof for k in that subtree. There will be $O(|V| \log n)$ such subtrees (and thus frontier proofs).
- For each of the $\log(n) - t$ remaining PCTs where k has no values, a frontier proof to prove k is not there. This is at most $O(\log n)$ frontier proofs.

Thus, the lookup proof size is $O\left(|V| \log n + f(|V| \log n + \log n)\right)$, where f is the frontier proof size. If instantiated with RSA accumulators, we call this scheme $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{rsa}}$, and it has $O(|V| \log n)$ lookup proofs. If instantiated with bilinear accumulators, we call it $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$, and it has $O(|V| \log^2 n)$ lookup proofs.

6.3.2 AADs with Less Space Overhead

Here, we take a different approach where PCTs accumulate both keys and their values in a manner that allows us to construct lookup proofs. Specifically, we increase the size of the domain of the underlying AAS from 2λ bits to 4λ bits so as to account for the value v, as depicted in Figure 6-2. That is, (k, v) would be inserted in the AAS as $k|v$, using the same algorithms from Section 5.3.5. This increases the trie height, making appends slower since more prefixes have to be accumulated. We call the RSA-based construction $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$ and the bilinear-based construction $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$.

Note that $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ has double the q-PKE parameters public parameters of $\mathsf{AAS}_{\mathsf{bilinear}}$ (and of $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$). Specifically, the server needs $q = 4\lambda n + 1$ and clients need $q = 4\lambda + 1$. On the other hand, $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ and $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$ have lower space overhead than $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$ and $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$: $O(\lambda n)$ versus $O(\lambda n \log n)$.

Supporting Large Domains and Multisets. To handle keys and values longer than 2λ bits, we store $\mathcal{H}(k)|\mathcal{H}(v)$ in the AAD (rather than $k|v$), where $\mathcal{H}$ is a CRHF and we can retrieve the actual value v from another repository. To support multisets (same v can be inserted twice for a k), the server can insert $\mathcal{H}(\mathcal{H}(v)|i)$ for the ith occurrence of (k, v).

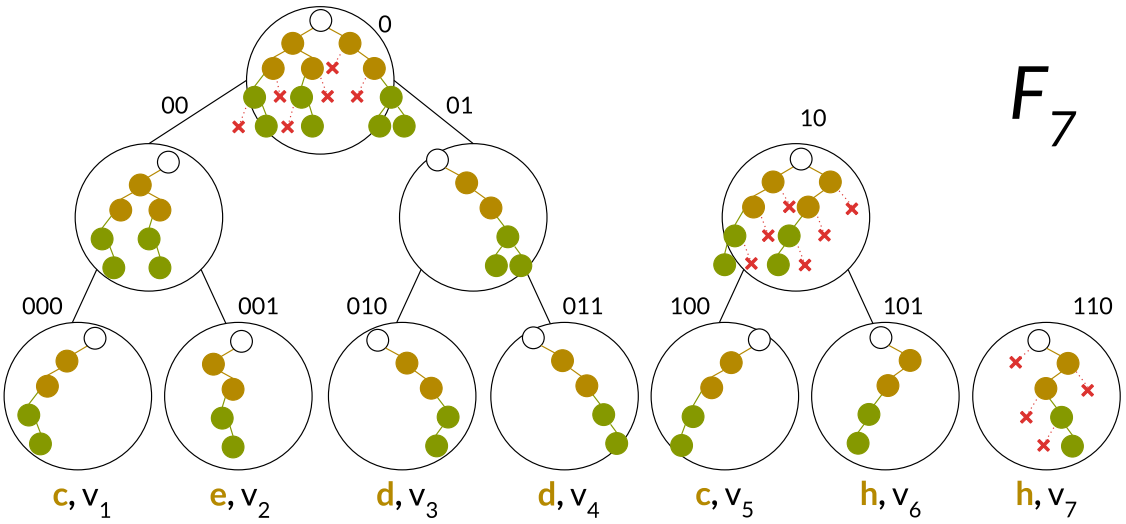


Figure 6-2: A dynamic AAD for dictionary $\{(c, v_1), (c, v_5), (d, v_3), (d, v_4), (e, v_2), (h, v_6), (h, v_7)\}$ with $\lambda = 1$. The tries in this AAD are built not just over keys but also over values, unlike our AAS from Figure 5-5 and unlike our "short" AAD from Figure 6-1. Specifically, the first 2λ edges in a trie path encode the hash of the key and the last 2λ edges encode the hash of the value. As in our AAS, *only root nodes* in the forest store a Frontier Communion Tree (FCT). These root FCTs are used to "provably-guide" a search for all the values of a key.

6.3.2.1 Proving Lookups

First, consider a key k with no values. A lookup proof consists of $O(\log n)$ frontier proofs, one for each PCT in the forest where a prefix of k is missing. This is very similar to a non-membership proof in the AAS (see Section 5.3).

But what if k has one or more values? First, the lookup proof contains paths to PCT leaves with k's values (i.e., with elements of the form $k|v$), much like a membership proof in an AAS. But what is to guarantee *completeness* of the response? What if a malicious server leaves out one of the values of key k? (This is important in transparency logs where users look up their own PKs and must receive all of them to detect impersonation attacks.)

Lower Frontiers. We use the same frontier technique as in the AAS to convince clients values are not being left out. Specifically, the server proves specific prefixes for the *missing values* of k are not in the PCTs (and thus are not maliciously being left out). This is best illustrated with the example in Figure 6-3 where the security parameter is $\lambda = 2$.

Suppose the server wants to prove $k = 0000$ has *complete* set of values $V = \{v_1 = 0001, v_2 = 0011\}$. Consider a trie over $k|v_1$ and $k|v_2$ and note that $F_V^{[k]} = \{(0000|1), (0000|01), (0000|0000), (0000|0010)\}$ is the set of all frontier prefixes for the missing values of k. We call this set the *lower frontier* of k relative to V. The key idea for showing completeness of V is to *prove all these lower frontier prefixes are in the FCT* via frontier proofs (as defined in Section 5.3). Since there are $O(\lambda)$ lower frontier prefixes, one for each value $v \in V$ of key k, the server will send $O(\lambda|V|)$ frontier proofs. Thus, a proof for a key k with a single value v (i.e., $V = \{v\}$) consists of:

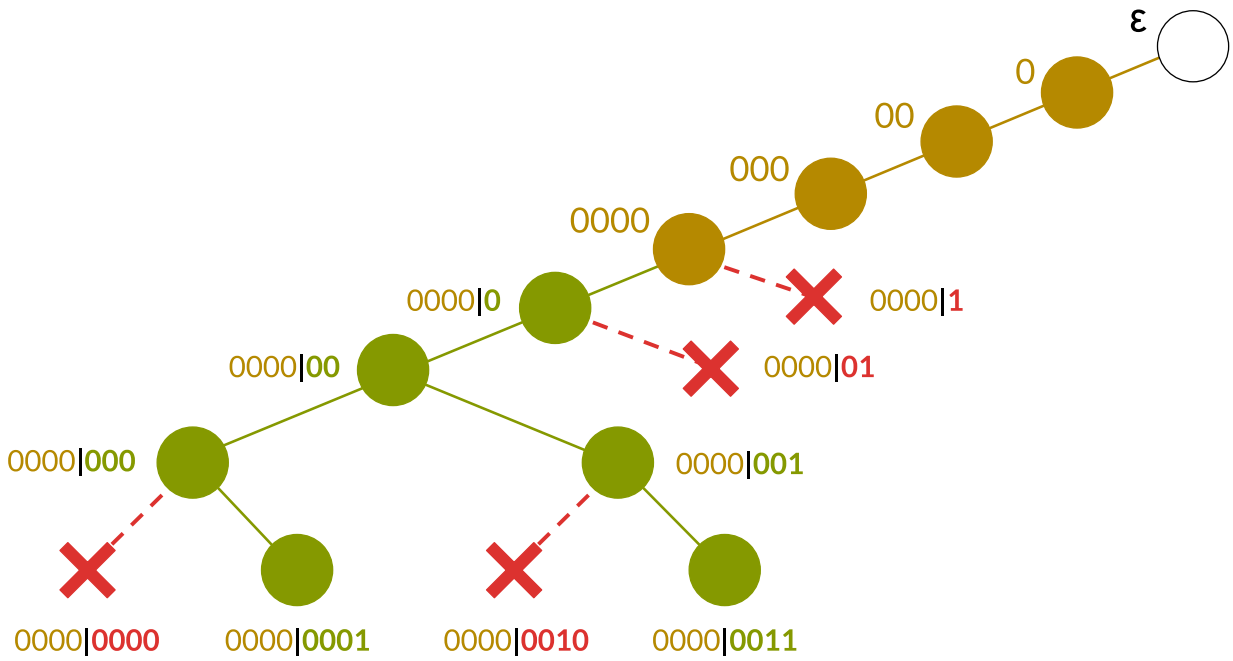


Figure 6-3: A "tall" trie for a key k with hash 0000 that has two values with hashes 0001 and 0011, respectively. The security parameter is $\lambda = 2$. The lower frontier consists of the red nodes in the lower half of the trie, which correspond to missing values for k.

- A proof path to (k, v) in some PCT (to show v is one of k's values),
- $O(\lambda)$ frontier proofs in the FCT corresponding to that PCT, one for each prefix in the lower frontier $F^{[k]}_{\{v\}}$ (to guarantee completeness of $\{v\}$ in that PCT),
- A frontier proof for a missing prefix of k in each one of the remaining $O(\log n)$ FCTs (to prove k has no values there).

Keys with Multiple Values. Let us generalize lookup proofs to keys with many values. Assume the key k has (zero or more) values in V that are "spread out" across $t \geq 0$ of the $\leq \log n$ PCTs in the forest. For PCTs with no values for k, the server proves non-membership of k in the corresponding FCT via one frontier proof. For the ith PCT, where k has values $v \in V_i$, the server proves the completeness of V_i by giving $O(\lambda|V_i|)$ frontier proofs, one for each lower frontier prefix. Thus, the lookup proof consists of:

- For each value $v \in V$, a PCT proof path to (k, v) in the forest. These add $O(|V| \log n)$ overhead.
- Across the t PCTs where k has values, $O(\lambda|V|)$ frontier proofs will be sent.
- For each of the $\log(n) - t$ remaining PCTs where k has no values, a frontier proof in its corresponding FCT to prove k is not there.

Thus, the total number of frontier proofs is $O(\lambda|V| - t + \log n) = O(\lambda|V| + \log n)$. This means the lookup proof size is $O(f(\lambda|V| + \log n) + |V| \log n)$-sized, where f is the construction-specific frontier proof size. Next, we describe how to remove the λ factor in the

Scheme	Space	Public params	Append time	Lookup proof size	Inclusion proof size	Append-only proof size
$\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$	λn	$4\lambda n$	$\lambda \log^3 n$	$(\lvert V\rvert + \log n)\log n$	$\lvert V\rvert \log n$	$\log n$
$\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$	$\lambda n \log n$	$2\lambda n$	$\lambda \log^3 n$	$\lvert V\rvert \log^2 n$	$\lvert V\rvert \log n$	$\log n$
$\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$	λn	1	$\lambda \log^3 n \log\log n$	$\lvert V\rvert \log n$	$\lvert V\rvert$	$\log n$
$\mathsf{AAD}^{\mathsf{short}}_{\mathsf{rsa}}$	$\lambda n \log n$	1	$\lambda \log^3 n \log\log n$	$\lvert V\rvert \log n$	$\lvert V\rvert$	$\log n$

Table 6.1: Complexity of our AAD schemes. n is the number of key-value pairs in the AAD. Lookup proofs are for keys with $|V|$ values. While our "short" constructions require more space, they speed up appends due to their "shorter" tries.

expression above.

Smaller Lookup Proofs in $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$. Since frontier proofs are $O(\log n)$-sized, the $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ lookup proof size will be $O(\lambda|V|\log n + \log^2 n)$. We show how to decrease this to $O(|V|\log n + \log^2 n)$. We begin with the case where k has one value.

We know from before that a lookup proof for $(k, \{v\})$ is $O(\lambda \log n)$-sized (since the $\lambda \log n$ term dominates the $\log^2 n$ term). Note that the $O(\lambda)$ overhead comes from having to prove that all $O(\lambda)$ lower frontier prefixes of k (relative to V) are in an FCT. The key idea is to *group all these lower frontier prefixes* into a single FCT leaf, creating a frontier accumulator over all of them. As a result, instead of having to send $O(\lambda)$ frontier proofs (one for each lower frontier prefix), we send a single $O(\log n)$-sized frontier proof for a single FCT leaf which contains all $O(\lambda)$ lower frontier prefixes of k relative to $\{v\}$.

We can generalize this idea. Specifically, if k has $|V_i|$ values in the ith FCT in the forest, then k's lower frontier relative to V_i has $O(\lambda|V_i|)$ prefixes. Then, for each FCT i, we split the lower frontier prefixes of k associated with V_i into separate FCT leaves each of size at most $4\lambda + 1$. We remind the reader that clients have enough public parameters to reconstruct the accumulators in these FCT leaves and verify the frontier proof. As a result, the lookup proof size becomes $O(|V|\log n + \log^2 n)$.

Smaller Lookup Proofs in $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$. Because frontier proofs are constant-sized in $\mathsf{AAS}_{\mathsf{rsa}}$, the $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$ lookup proof size will be $O(\lambda|V| + \log n + |V|\log n)$. The same key idea from above can be used to bring the proof size down to $O(|V| + \log n + |V|\log n) = O(|V|\log n)$. Recall that, in $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{rsa}}$, constant-sized membership witnesses are computed for every frontier prefix. Thus, in each FCT, for each key k's lower frontier prefixes, we can aggregate its membership witnesses in batches of size $4\lambda + 1$ using the technique from [31].

6.3.3 Supporting Inclusion Proofs

Another useful proof for a transparency log is an *inclusion proof* which only returns *one of the values* of key k (while lookup proofs return *all* values of a key k). For example, in Certificate Transparency (CT), browsers are supposed to verify an inclusion proof of a website's certificate before using it. Our AADs support inclusion proofs too. They consist

of a path to a PCT leaf with the desired key-value pair. Since they do not require frontier proofs, inclusion proofs are only $O(\log n)$-sized.

6.3.4 Asymptotic Analysis

We have already analyzed the lookup proof sizes of each construction in Sections 6.3.1.1 and 6.3.2.1. The complexity analyses for the bilinear- and RSA-based AADs are similar to the ones from Section 5.3.4 and Section 5.4.3, respectively. To avoid repetition, we give the AAD complexities in Table 6.1.

6.4 Bilinear-based AAD Implementation

In this section, we evaluate our $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ (not AAS) construction's proof sizes, append times and memory usage. We find that append times and memory usage are too high for a practical deployment and discuss how they might be improved in future work (see Sections 6.4.1.1 and 6.4.1.4). If they are improved, we find AADs can save bandwidth relative to CT and CONIKS and we describe exactly when and how much in Section 6.4.2.1.

Codebase and Testbed. We implemented our *amortized* AAD construction from Chapter 6 in 5700 lines of C++. Its *worst-case* append time is $O(\lambda n \log^2 n)$ while its amortized append time is $O(\lambda \log^3 n)$. We used Zcash's `libff` [189] as our elliptic curve library with support for a 254-bit Barretto-Naehrig curve with a Type III pairing [15]. We used `libfqfft` [190] to multiply polynomials and `libntl` [193] to divide polynomials and compute GCDs. Our code is available at:

`https://github.com/alinush/libaad-ccs2019`.

We ran our evaluation in the cloud on Amazon Web Services (AWS) on a r4.16xlarge instance type with 488 GiB of RAM and 64 VCPUs, running Ubuntu 16.04.4 (64-bit). This instance type is "memory-optimized" which, according to AWS, means it is "designed to deliver fast performance for workloads that process large data sets in memory."

6.4.1 Microbenchmarks

In this subsection, we measure:

- The average time to append a key-value pair,
- The size of lookup proofs,
- The size of append-only proofs,
- Our memory usage.

6.4.1.1 Append Times

Starting with an empty AAD, we append key-value pairs to it and keep track of the *cumulative average append-time*. Recall that appends are amortized in our construction (but can be

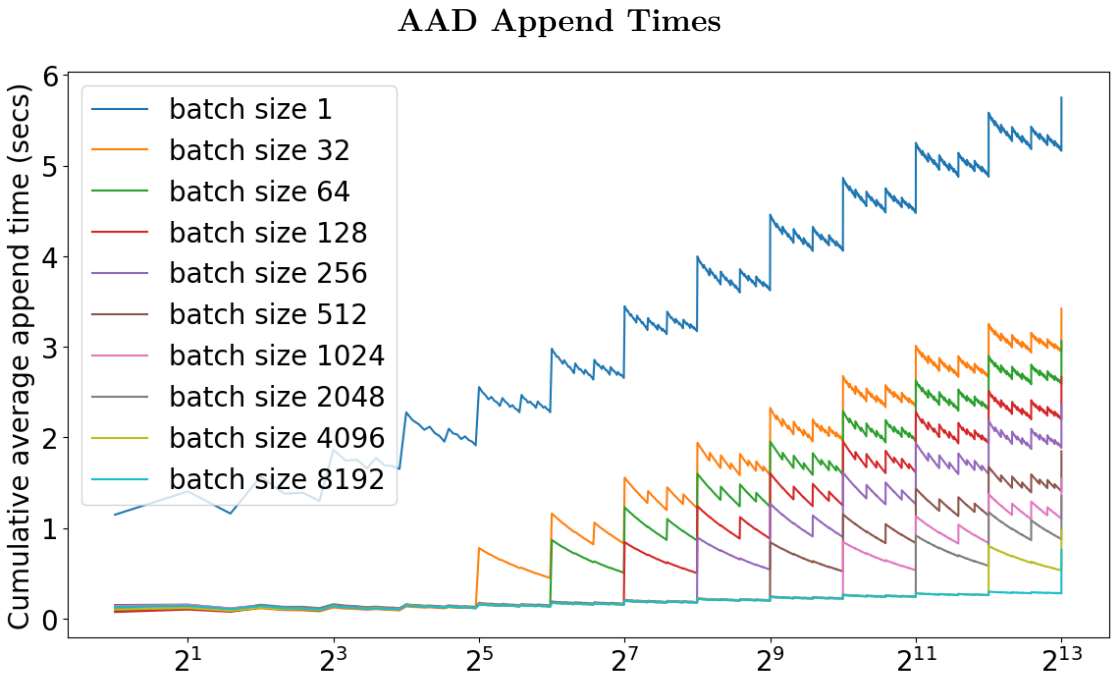
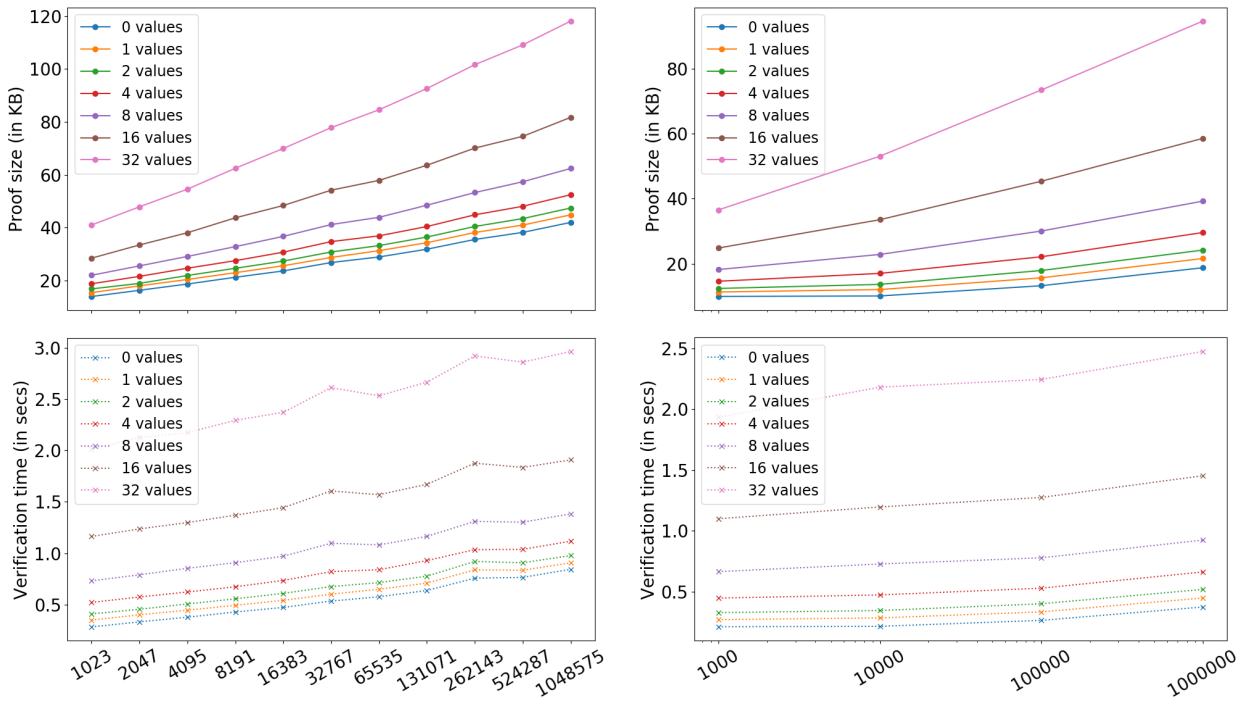


Figure 6-4: This graph plots the average append-time (y-axis) as measured after n appends (x-axis). "Spikes" occur when two PCTs of size b are merged in the forest (see Figure 5-4), which triggers a new FCT computation, where b is the batch size. The average append time increases with the dictionary size since bigger PCTs are being created and merged (and bigger FCTs are being computed). Bigger batch size means PCTs are merged less frequently and FCTs are created less frequently, which decreases the average append time.

de-amortized using known techniques [165,166]). As a result, in our benchmark some appends are very fast (e.g., 25 milliseconds) while others are painfully slow (e.g., 1.5 hours). To keep the running time of our benchmark reasonable, we only benchmarked $2^{13} = 8192$ appends. We also investigate the effect of batching on append times. Batching $k = 2^i$ appends together means we only compute one FCT for the full tree of size k created after inserting the batch. In contrast, without batching, we would compute k FCTs, one for each new forest root created after an append. Figure 6-4 shows that the average append time is 5.75 seconds with no batching and 0.76 seconds with batch size 8192. (For batch sizes $32, 64, \ldots, 4096$, the average times per append in milliseconds are 3422, 3064, 2644, 2361, 1848, 1548, 1353 and 976, respectively.) These times should increase by around 3.5 seconds if we benchmarked 2^{20} appends.

Speeding Up Appends. The bottleneck for appends is computing the FCTs. Although we used `libff`'s multi-threaded multi-exponentiation to compute accumulators faster, there are other ways to speed up appends that we have not explored. First, we can parallelize computing (1) the polynomials on the same level in a FCT, (2) the smaller accumulators at lower levels of the FCT, where multi-threaded multi-exponentiation does not help as much and (3) the subset witnesses in the forest. Second, we can reuse some of the previously computed accumulators when computing a new FCT. Third, our PCT and FCT constructions



(a) Worst-case AAD sizes

(b) Average-case AAD sizes

Figure 6-5: Lookup proof sizes and verification times (y-axis) increase with the dictionary size (x-axis) and with the number of values the proof attests for. Figure 6-5a tells the same story as Figure 6-5b, except the proof sizes and verification times are slightly higher since these *worst-case* AADs have more trees in the forest.

require "extractable" counterparts of the accumulators, which almost triple the time to commit to a polynomial. We hope to remove this expensive requirement by proving our construction secure in the generic group model, similar to new SNARK constructions [112]. Finally, techniques for distributed FFT could speed up polynomial operations [212].

6.4.1.2 Lookup Proofs

We investigate three factors that affect lookup proof size and verification time: (1) the dictionary size, (2) the number of trees in the forest and (3) the number of values of a key. Our benchmark creates AADs of ever-increasing size n. For speed, instead of computing accumulators, we simply pick them uniformly at random. (Note that this does not affect the proof verification time.) We measure *average* proof sizes for keys with ℓ values in an AAD of size n, where $\ell \in \{0, 1, 2, 4, 8, 16, 32\}$. (Recall that a key with ℓ values requires ℓ frontier proofs.)

To get an average, for every ℓ, we set up 10 different *target* keys so each key has ℓ values. The rest of the inserted keys are random (and simply ignored by the benchmark). Importantly, we randomly disperse the target key-value pairs throughout the forest to avoid having all values of a key end up in consecutive forest leaves, which would artificially decrease

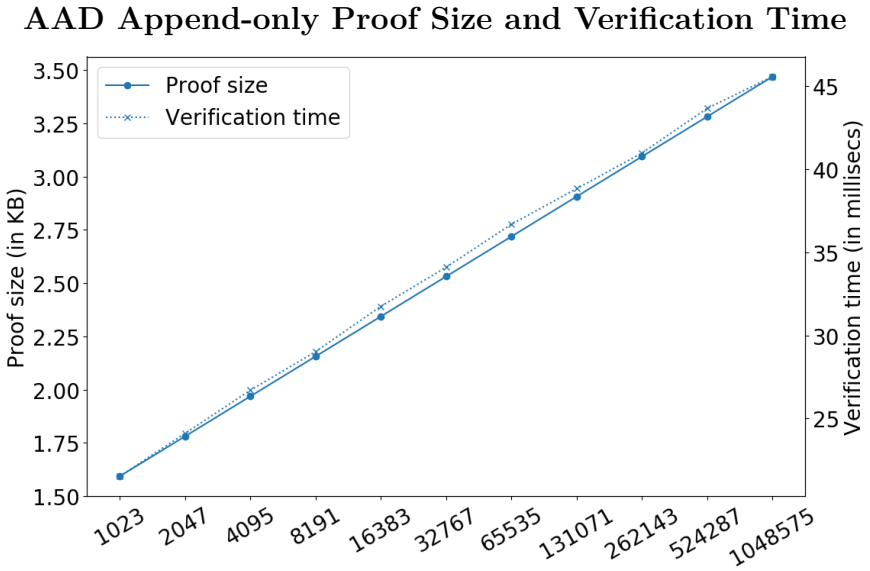


Figure 6-6: This graph shows that our append-only proof sizes and verification times (y-axis) are quite small and scale logarithmically with the dictionary size (x-axis).

the proof size. Once the dictionary reaches size n, we go through every target key with ℓ values, compute its lookup proof, and measure the size and verification time. Then, for each ℓ, we take an average over its 10 target keys. We repeat the experiment for increasing dictionary sizes n and summarize the numbers in Figures 6-5a and 6-5b. Proof verification is single-threaded.

Worst-Case vs. Best-Case Dictionary Sizes. Recall that some dictionary sizes are "better" than others because they have fewer trees in the forest. For example, a dictionary of (worst-case) size $2^i - 1$ will have i trees in the forest and thus i FCTs. Thus, a lookup proof must include frontier proofs in all i FCTs. In contrast, a dictionary of size 2^i only has a single tree in the forest, so a lookup proof needs only one frontier proof. Indeed, our evaluation shows that lookup proofs are smaller in AADs of size 10^i (see Figure 6-5b) compared to $2^i - 1$ (see Figure 6-5a). For example, for a key with 32 values, the proof averages 95 KiB for size 10^6 and 118 KiB for size $2^{20} - 1$.

6.4.1.3 Append-only Proofs

This benchmark appends random key-value pairs until it reaches a target size $n = 2^{i+1} - 1$. Then, it measures the append-only proof size (and verification time) between AADs of size n and $m = 2^i - 1$. We benchmarked on $2^i - 1$ AAD sizes to illustrate worst-case $\Theta(i)$ append-only proof sizes. To speed up the benchmark, we randomly pick accumulators in the forest. Append-only proof verification is single-threaded. Our results show append-only proofs are reasonably small and fast to verify (see Figure 6-6). For example, the append-only proof between sizes $2^{19} - 1$ and $2^{20} - 1$ is 3.5 KiB and verifies in 45 milliseconds.

6.4.1.4 Memory Usage

Our lookup proof benchmark was the most memory-hungry: it consumed 263 GiB of RAM for an AAD of size $n = 2^{20} - 1$. In contrast, the append-only proof benchmark consumed only 12.5 GiBs of memory, since it did not need FCTs. As an example, when $n = 2^{20} - 1$, all FCTs combined have no more than $390n$ nodes. Since we are using Type III pairings, each node stores three accumulators (two in $\mathbb{G}_1$ and one in $\mathbb{G}_2$) in 384 bytes (due to `libff`'s $3\times$ overhead). Thus, the FCT accumulators require no more than 147 GiB of RAM. The rest of the overhead comes from our pointer-based FCT implementation and other bookkeeping (e.g., polynomials). The q-PKE public parameters could have added 64 GiBs of RAM, but these two benchmarks did not need them.

Improving Memory. A new security proof could eliminate the additional $\mathbb{G}_1$ and $\mathbb{G}_2$ accumulators and reduce FCT memory by $2.66\times$ and the size of the public parameters by $1.33\times$ (see Section 6.4.1.1). A more efficient representation of group elements than `libff`'s could also reduce FCT memory by $3\times$. An efficient array-based implementation of PCTs and FCTs could further reduce memory by tens of gigabytes. Finally, the $390\times$ overhead of FCTs can be drastically reduced by carefully grouping upper frontier prefixes together in a FCT leaf, similar to the grouping of lower frontier prefixes from Chapter 6. However, doing this without increasing the lookup proof size too much remains to be investigated.

6.4.2 Comparison to Merkle Tree Approaches

How do AADs compare to Merkle prefix trees or History Trees (HTs), which are used in CONIKS and Certificate Transparency (CT), respectively? First of all, appends in AADs are orders of magnitude slower because of the overheads of cryptographic accumulators and remain to be improved in future work (see Section 6.4.1.1).

Lookup proofs in prefix trees are much smaller than in AADs. In a prefix tree of size 2^{20}, a proof consisting of a Merkle path would be around 640 bytes. In comparison, our proofs for a key with 32 values are 152 times to 189 times more expensive (depending on the number of trees in the forest). On the other hand, append-only proofs in AADs are $O(\log n)$, much smaller than the $O(n)$ in prefix trees. For example, our Golang implementation of prefix trees, shows that the append-only proof between trees of size 2^{19} and 2^{20} is 32 MiB (as opposed to 3.5 KiB in AADs). The proof gets a bit smaller when the size gap between the dictionaries is larger but not by much: 14.6 MiB between 10^5 and 10^6.

Lookup proofs in history trees (HTs) are $O(n)$-sized, compared to $O(\log^2 n)$ in AADs. This is because, to guarantee completeness, the HT proof must consist of all key-value pairs. On the other hand, append-only proofs in AADs are slightly larger than in HTs. While our proofs contain approximately the same number of nodes as in HT proofs, our nodes store two prefix accumulators in $\mathbb{G}_1$ and a subset witness in $\mathbb{G}_2$ (in addition to a Merkle hash). This increases the per-node proof size from 32 bytes to $32 + 64 + 64 = 160$ bytes.

6.4.2.1 When do AADs Reduce Bandwidth?

Asymptotically, AAD proof sizes outperform previous work. But in practice, our evaluation shows AAD proof sizes are still larger than ideal, especially lookup proofs. This begs the question: *In what settings do AADs reduce bandwidth in transparency logs?* We answer this question below while acknowledging that AAD append times and memory usage are not yet sufficiently fast for a practical deployment (see Section 6.4.1.1).

To begin, consider a key transparency log with approximately one billion entries (i.e., an entry is a user ID and its PK).

CONIKS. In a CONIKS log, each user must check their PK in every digest published by the log server. Let D denote the number of digests published per day by the log server. This means the CONIKS log server will, on average, send $960 \cdot \mathsf{D}$ bytes per day per user (without accounting for the overhead of VRFs [153] in CONIKS). If this were an AAD log, then each user (1) gets the most recent digest via an append-only proof and (2) checks their PK only in this digest via a lookup proof.

Let C denote the number of times per day a user checks his PK (and note that, in CONIKS, $C = D$). Since the lookup proof is for the user's PKs not having changed, it only needs to contain frontier proofs. Extrapolating from Figure 6-5b, such an average-case lookup proof is 40 KiB (in an AAD of size one billion). Similarly, an append-only proof would be 7 KiB. This means the AAD log server will, on average, send $47 \cdot 1024 \cdot C$ bytes per day per user. Thus, AADs are more efficient when $.0199 \cdot D/C > 1$.

In other words, AADs will be more bandwidth-efficient in settings where log digests must be published frequently (i.e., D is high) but users check their PK sporadically (i.e., C is low). For example, if $D = 250$ (i.e., a new digest every 6 minutes) and $C = 0.5$ (i.e., users check once every two days), then AADs result in $10\times$ less bandwidth.

Certificate Transparency (CT). Recall that CT lacks succinct lookup proofs. As a result, domains usually trust one or more *monitors* to download the log, index it and correctly answer lookup queries. Alternatively, a domain can act as a monitor itself and keep up with every update to the log. We call such domains *monitoring domains*.

Currently, CT receives 12.37 certificates per second on average [109], with a mean size of 1.4 KiB each [82]. Thus, a CT log server will, on average, send $12.37 \cdot 1.4 \cdot 1024 = 17,733.63$ bytes per second per monitoring domain. In contrast, AADs require $47 \cdot 1024 \cdot C/86,400 = .557 \cdot C$ bytes per second per monitoring domain. As before, C denotes how many times per day a monitoring domain will check its PK in the log. Thus, AADs are more efficient when $31,837/C > 1$.

So even if domains monitor very frequently (e.g., $C = 100$), AADs are more bandwidth efficient. However, we stress that our append times and memory usage must be reduced for a practical deployment to achieve these bandwidth savings (see Sections 6.4.1.1 and 6.4.1.4).

Part II

Threshold Cryptosystems

Chapter 7

Introduction

Due to the popularity of cryptocurrencies, interest in Byzantine fault tolerant (BFT) systems has been steadily increasing [13,72,102,105,113,130,155,198,211]. At the core of BFT systems often lie simpler threshold cryptosystems such as threshold signature schemes (TSS) [28,192], verifiable secret sharing (VSS) protocols [61,128,173] and distributed key generation (DKG) protocols [99,126,172]. For example, TSS and DKG protocols are used to scale consensus protocols [46,105,113]. Furthermore, DKG protocols [99] are used to securely generate keys for TSS [125], to generate nonces for interactive TSS [96,196], and to build proactively-secure threshold cryptosystems [117,118]. Finally, VSS is used to build multi-party computation (MPC) protocols [100], random beacons [52,130,198] and is the key component of DKG protocols.

Despite their usefulness, TSS, VSS and DKG protocols do not scale well in important settings. For example, BFT systems often operate in the *honest majority* setting, with n total players where $t > n/2$ players must be honest. In this setting, *t-out-of-n threshold cryptosystems*, such as TSS, VSS and DKG, require time quadratic in n [28,89,128,173]. This is because of two reasons. First, reconstruction of secrets, a key step in any threshold cryptosystem, is typically implemented naively using $\Theta(t^2)$ time polynomial interpolation, even though faster algorithms exist [207]. This makes aggregating threshold signatures and reconstructing VSS or DKG secrets slow for large t. Second, either the dealing round, the verification round or the reconstruction phase in VSS and DKG protocols require $\Theta(nt)$ time. Fundamentally, this is because current polynomial commitment schemes require $\Theta(nt)$ time to either compute or verify all proofs [89,128,173]. In this thesis, we address both of these problems.

Contributions. Our first contribution is a BLS TSS [28] with $\Theta(t \log^2 t)$ aggregation time, $\Theta(1)$ signing and verification times and $\Theta(1)$ signature size (see Section 10.1). In contrast, previous schemes had $\Theta(t^2)$ aggregation time (see Section 7.1.1). We implement our fast BLS TSS in C++ and show it outperforms the naive BLS TSS as early as $n \geq 511$ and scales to n as large as 2 million (see Section 10.4.1). At that scale, we can aggregate a signature $3000\times$ faster in 46 seconds compared to 1.5 days if done naively. Our fast BLS TSS leverages a $\Theta(t \log^2 t)$ time *fast Lagrange* interpolation algorithm [207], which outperforms the $\Theta(t^2)$ time *naive Lagrange* algorithm.

Our second contribution is a space-time trade-off for computing evaluation proofs in *KZG*

Scheme	Dealing round time	Verification round time	Complaint round time	Reconstruct time (no interpol.)	Dealing communic. (broadcast)	Dealing communic. (private)
Feldman VSS [89]	$n \log n$	t	t^2	nt	t	n
JF-DKG [99]	$n \log n$	nt	t^3	nt	t	n
eVSS [128]	nt	1	t	n	1	n
eJF-DKG [126]	nt	n	t^2	n	1	n
AMT VSS	$n \log t$	$\log t$	$t \log t$	$n \log t$	1	$n \log t$
AMT DKG	$n \log t$	$n \log t$	$t^2 \log t$	$n \log t$	1	$n \log t$

Table 7.1: Per-player *worst-case* asymptotic complexity of (t, n) VSS/DKG protocols.

polynomial commitments [128] (see Chapter 9). KZG commitments are quite powerful in that their size and the time to verify an evaluation proof are both constant and do not depend on the degree of the committed polynomial. We show how to compute n evaluation proofs on a degree t polynomial in $\Theta(n \log t)$ time. Each proof is of size $\lfloor \log t \rfloor - 1$ group elements. Previously, each proof was just one group element but computing all proofs required $\Theta(nt)$ time. Our key technique is to authenticate a polynomial multipoint evaluation at the first n roots of unity (see Section 2.2), obtaining an *authenticated multipoint evaluation tree (AMT)*. Importantly, similar to KZG proofs, our AMT proofs remain homomorphic (see Section 9.1.3), which is useful when we apply them to distributed key generation (DKG) protocols. Finally, AMTs give rise to a new *vector commitment (VC)* scheme [54, 58] as we discuss in Section 9.2.

Our third contribution is AMT VSS, a scalable VSS with a $\Theta(n \log t)$ time sharing phase, an $O(t \log^2 t + n \log t)$ time reconstruction phase, $\Theta(1)$-sized broadcast (during dealing round) and $\Theta(n \log t)$ overall communication. AMT VSS improves over previous VSS protocols which, in the worst case, incur $\Theta(nt)$ computation. However, this improvement comes at the cost of slightly higher verification times and communication (see Table 7.1). Nonetheless, in Section 10.4, we show AMT VSS outperforms eVSS [128], the most communication-efficient VSS, as early as $n = 63$. Importantly, AMT VSS is highly scalable. For example, for $n \approx 2^{17}$, we reduce the best-case end-to-end time of eVSS from 2.2 days to 8 minutes.

Our fourth contribution is AMT DKG, a DKG with a $\Theta(n \log t)$ time sharing phase (except for its quadratic time complaint round), an $O(t \log^2 t + n \log t)$ time reconstruction phase, a $\Theta(1)$-sized broadcast (during dealing round) and $\Theta(n \log t)$ per-player dealing communication. AMT DKG improves over previous DKGs which, in the worst case, incur $\Omega(nt)$ computation. Once again, this improvement comes at the cost of slightly higher verification times and communication (see Table 7.1). Nonetheless, in Section 10.4, we show AMT DKG outperforms eJF-DKG [126], the most communication-efficient DKG, as early as $n = 63$. For $n \approx 2^{17}$, we reduce the best-case end-to-end time of eJF-DKG from 2.4 days to 4 minutes.

Our last contribution is an open-source implementation:

`https://github.com/alinush/libpolycrypto/`

Limitations. Our work only addresses TSS, VSS and DKG protocols secure against *static* adversaries. However, *adaptive security* can be obtained, albeit with some overheads [1, 50, 89, 92, 122]. We only target *synchronous* VSS and DKG protocols, which make strong assumptions about the delivery of messages. However, recent work [185] shows how to

instantiate such protocols using the Ethereum blockchain [211]. Our VSS and DKG protocols require a *trusted setup* (see Section 2.6). Our evaluation only measures the computation in VSS and DKG protocols and does not measure network delays that would arise in a full implementation on a real network. Our techniques slightly increase the communication overhead of VSS and DKG protocols from $\Theta(n)$ to $\Theta(n \log t)$. However, when accounting for the time savings, the extra communication is worth it. Still, we acknowledge communication is more expensive than computation in some settings. Finally, we do not address the worst-case quadratic overhead of complaints in DKG protocols. We leave scaling this to future work.

7.1 Related Work

7.1.1 Threshold Signature Schemes (TSS)

Threshold signatures and threshold encryption were first conceptualized by Desmedt [73], who proposed inefficient constructions based on generic multi-party computation (MPC) protocols [106]. Desmedt and Frankel later efficiently instantiated these ideas via a threshold-variant of ElGamal encryption [74]. Since then, many threshold signatures based on *Shamir secret sharing* (see Section 8.4) have been proposed [28, 75, 96, 97, 114, 171, 192, 196]. To the best of our knowledge, none of these schemes addressed the $\Theta(t^2)$ time required for polynomial interpolation. Furthermore, all current BLS TSS [28] implementations seem to use this quadratic algorithm [59, 77, 154, 204] and thus do not scale to large t. In contrast, our work uses $\Theta(t \log^2 t)$ fast Lagrange interpolation and scales to $t = 2^{20}$ (see Section 10.1).

An alternative to a TSS is a *multi-signature scheme (MSS)*. Unlike a TSS, an MSS does not have a unique, constant-sized public key (PK) against which all final signatures can be verified. Instead, the PK is dynamically computed given the contributing signers' IDs and their public keys. This means that a t-out-of-n MSS must include the t signer IDs as part of the signature, which makes it $\Omega(t)$-sized. Furthermore, MSS verifiers must have all signers' PKs, which are of $\Omega(n)$ size. To fix this, the PKs can be Merkle-hashed but this now requires including the PKs and their Merkle proofs as part of the MSS [197]. (This communication overhead can be addressed by using a more communication-efficient vector commitment scheme [54], but only by introducing additional computational overhead.) On the other hand, an MSS is much faster to aggregate than a TSS. Still, due to its $\Omega(t)$ size, an MSS does not always scale.

7.1.2 Verifiable Secret Sharing (VSS)

VSS protocols were introduced by Chor et al. [61]. Feldman proposed the first efficient, non-interactive VSS with computational hiding and information-theoretic binding [89]. Pedersen introduced its counterpart with information-theoretic hiding and computational binding [173]. Both schemes require a $\Theta(t)$-sized broadcast during dealing.

Kate et al.'s eVSS reduced this to $\Theta(1)$ using constant-sized polynomial commitments [128]. eVSS also reduced the verification round time from $\Theta(t)$ to $\Theta(1)$. However, eVSS's $\Theta(nt)$ dealing time scales poorly when $t \approx n$. Our work improves eVSS to $\Theta(n \log t)$ dealing time at the cost of $\log t$ verification round time. We also increase communication from $\Theta(n)$ to

$n \log t$ (see Table 7.1).

7.1.3 Publicly-Verifiable Secret Sharing (PVSS)

Stadler proposed publicly verifiable secret sharing (PVSS) protocols [195] where any *external verifier* can verify the VSS protocol execution. As a result, PVSS is less concerned with players individually and efficiently verifying their shares, instead enabling external verifiers to verify all players' (encrypted) shares. Schoenmakers proposed an efficient (t, n) PVSS protocol [187] where dealing is $\Theta(n \log n)$ time and external verification of all shares is $\Theta(nt)$ time, later improved to $\Theta(n)$ time by Cascudo and David [52]. Unfortunately, when the dealer is malicious, PVSS still needs $\Theta(nt)$ computation during reconstruction. Furthermore, PVSS might not be a good fit in protocols with a large number of players. In this setting, it might be better to base security on a *large*, threshold number of honest players who individually and efficiently verify their own share rather than on a *small* number of external verifiers who must each do $\Omega(n)$ work. Indeed, recent work explores the use of VSS within BFT protocols *without* external verifiers [17]. Nonetheless, our AMT VSS protocol can be easily modified into a PVSS since an AMT for all n proofs can be batch-verified in $\Theta(n)$ time (see Section 10.2.3).

7.1.4 Distributed Key Generation (DKG)

DKG protocols were introduced by Ingemarsson and Simmons [121] and subsequently improved by Pedersen [172,173]. Gennaro et al. [99] noticed that if players in Pedersen's DKG refuse to deal [172], they cannot be provably blamed and fixed this in their new JF-DKG protocol. They also showed that secrets produced by Pedersen's DKG can be *biased*, and fixed this in their New-DKG protocol. Neji et al. gave a more efficient way of debiasing Pedersen's DKG [159]. Gennaro et al. also introduced the first "fast-track" or optimistic DKG [100]. Canetti et al. modified New-DKG into an adaptively-secure DKG [50]. Fouque and Stern [91] gave a one-round DKG that removes the need for an expensive *complaint round* (see Algorithm 8). So far, all DKGs required a $\Theta(t)$-sized broadcast by each player.

Kate's eJF-DKG [126] reduced the dealer's broadcast to $\Theta(1)$ via constant-sized polynomial commitments [128], taking a first step towards scalability. eJF-DKG also reduced verification time from $\Theta(nt)$ per player to $\Theta(n)$ but at the cost of $\Theta(nt)$ dealing time per player. To summarize, all DKGs so far require $\Theta(nt)$ computation per player (in the worst case), while our AMT DKG requires $\Theta(n \log t)$.

So far, all these DKGs assume a synchronous communication model between players, which can be difficult to instantiate. Recently, ETHDKG [185] surpasses this difficulty using Ethereum [211]. Kate et al. introduced *asynchronous* DKG protocols [125, 126] based on bivariate polynomials. We have not investigated if our techniques apply there.

7.1.5 Polylogarithmic DKG

Canny and Sorkin present a polylogarithmic time DKG [51], a beautiful result that unfortunately has limitations. In certain settings, their protocol only requires $\Theta(\log^3 n)$ computation and communication per player. The key idea is that each player only talks to a *group* of $\log n$ other players, leading to a $\Theta(\log^3 n)$ per-player complexity. Unfortunately, their protocol

centralizes trust in a dealer who must "permute" the players before the protocol starts. The authors argue the dealer can be distributed amongst the players, but it is unclear how to do so securely while maintaining the $\Theta(\log^3 n)$ per player complexity.

Furthermore, their protocol does not efficiently support all thresholds (t, n). Instead, it only supports $((1/2 + \varepsilon)n, n)$ thresholds and tolerates $(1/2 - \varepsilon)n$ failures, where $\varepsilon \in (0, 1/2)$. Thus, their protocol can tolerate more failures only if ε is made very small. Unfortunately, a smaller ε causes the group size to increase, driving up the per-player complexity (see Appendix A.1). As a result, their protocol only scales in settings where a small fraction of failures is tolerated (e.g., 1/5) and a larger fraction of players is required to reconstruct (e.g., 4/5). Nonetheless, for their protocol to be truly distributed, the trusted dealer must be eliminated as a single point of failure.

7.1.6 DKG Implementations

Finally, the increasing popularity of BLS threshold signatures [28] has led to several DKG implementations. For example, recent works implement a DKG on top of the Ethereum blockchain [164, 176, 185]. Cryptocurrency companies such as DFINITY and GNOSIS implement a DKG as well [76, 104]. Finally, Distributed Privacy Guard (DKGPG) [116] implements a DKG for ElGamal threshold encryption [74] and for DSS threshold signatures [50]. All current implementations are based on Feldman [89] or Pedersen commitments [172] and require $\Theta(nt)$ time per player, which makes them difficult to scale.

Chapter 8

Preliminaries

8.1 Communication and Adversarial Model

8.1.1 Synchronous Communication

DKG and VSS protocols assume a *broadcast channel* for all actors to reliably communicate with each other [61, 172]. (In practice, this can be implemented using BFT protocols [185].) In addition, some protocols need *private and authenticated channels* between actors [89, 99, 126, 128, 172]. We focus on *synchronous* VSS and DKG protocols, where parties communicate in *rounds*. Within a round, each party performs some computation, (possibly) sends private messages to other players and broadcasts a message to everybody. By the end of the round, each party receives all messages sent in that round by other players (whether privately or via broadcast).

8.1.2 Static, Rushing, Threshold Adversaries

We assume computationally-bounded adversaries $\mathcal{A}$ that control up to $t-1$ players. We restrict ourselves to *static* $\mathcal{A}$'s who fix the set of $< t$ corrupted players before the protocol starts. We assume $\mathcal{A}$ can be *rushing* and can wait to hear all messages from all honest players in a round before privately sending or broadcasting his own message within that same round. The protocols in this thesis are *robust*: there are always t honest players who can reconstruct the secret. In the synchronous setting, robustness holds for all $t - 1 < n/2$ [99].

8.2 ℓ-Polynomial Diffie-Hellman (polyDH) Assumption

Definition 8.2.1 (ℓ-Polynomial Diffie-Hellman (polyDH) Assumption)**.** Given as input security parameter 1^λ, bilinear pairing parameters $\langle \mathbb{G}, \mathbb{G}_T, p, g, e\rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$ (see Definition 2.1.1), public parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(g;\tau) = \langle g, g^\tau, g^{\tau^2}, \ldots, g^{\tau^\ell}\rangle$ where $\ell = \mathsf{poly}(\lambda)$ and τ is chosen uniformly at random from $\mathbb{Z}_p^*$, no probabilistic polynomial-time adversary can output $(\phi(x), g^{\phi(\tau)}) \in \mathbb{Z}_p[X] \times \mathbb{G}$, such that $\ell < \deg\phi < 2^\lambda$, except with probability negligible in λ.

8.3 Threshold Signature Schemes (TSS)

A (t, n)-threshold signature scheme (TSS) is a protocol amongst n *signers* where *only* subsets of size $\geq t$ can produce a *digital signature* [182] on a message m. Many signature schemes can be turned into a TSS, such as RSA [182,192], Schnorr [98,186,196], ElGamal [84,97,171], ECDSA [96] and BLS [28, 34]. In this thesis, we focus on the BLS TSS because of its simplicity.

8.3.1 (Threshold) BLS signatures

A normal BLS signature on a message $m \in \{0,1\}^*$ is $\sigma = H(m)^s$ where $s \in_R \mathbb{F}_p$ is the *secret key* and $H : \{0,1\}^* \to \mathbb{G}$ is a hash function modeled as a random oracle. To verify the signature against the *public key* g^s, a bilinear map e is used to ensure that $e(H(m), g^s) \stackrel{?}{=} e(\sigma, g) \Leftrightarrow e(H(m), g)^s \stackrel{?}{=} e(H(m)^s, g)$.

To obtain a (t, n) BLS TSS [28], the secret key s is split amongst the n signers using (t, n) Shamir secret sharing (see Section 8.4). Specifically, each signer i has a *secret key share* s_i of s along with a *verification key* g^{s_i}. To produce a signature on m, each i computes a *signature share* $\sigma_i = H(m)^{s_i}$. Then, all σ_i's are sent to an *aggregator* (e.g., one of the signers).

Since some signers are malicious, their σ_i might not be valid. Thus, the aggregator verifies each σ_i by checking if $e(g^{s_i}, H(m)) \stackrel{?}{=} e(\sigma_i, g)$. (This works because σ_i is a normal BLS signature that should verify under g^{s_i}.) This way, the aggregator finds a subset T of t signers who produced a valid signature share σ_i. Now, the aggregator can compute the final signature as $\sigma = \prod_{i \in T} \sigma_i^{\mathcal{L}_i^T(0)} = H(m)^{\sum_{i \in T} s_i \mathcal{L}_i^T(0)} = H(m)^s$ via Lagrange interpolation (see Section 2.4). Importantly, aggregation never exposes the secret key s, which is interpolated "in the exponent." The time to *aggregate* the signature is $\Theta(t^2)$, dominated by the time to (naively) compute the $\mathcal{L}_i^T(0)$'s.

8.4 (Verifiable) Secret Sharing (VSS)

A (t, n) *secret sharing* scheme allows a *dealer* to split up a secret s amongst n *players* such that *only* subsets of size $\geq t$ players can reconstruct s. Secret sharing schemes were introduced independently by Shamir [191] and Blakley [27]. In this thesis, we focus on *Shamir's secret sharing (SSS)* protocol and its extensions.

SSS is split into two phases. In the *sharing phase*, the dealer picks a degree $t-1$, random, univariate polynomial ϕ, lets $s = \phi(0)$ and distributes a *share* $s_i = \phi(i)$ to each player $i \in [n]$. In the *reconstruction phase*, any subset $T \subset [n]$ of t honest players can reconstruct s by sending their shares to a *reconstructor*. (This can be one of the players, or another 3rd party.) For each $i \in T$, the reconstructor computes a *Lagrange coefficient* $\mathcal{L}_i^T(0) = \prod_{j \in T, j \neq i} \frac{0-j}{i-j}$. Then, he computes the secret as $s = \phi(0) = \sum_{i \in T} \mathcal{L}_i^T(0) s_i$ via Lagrange interpolation (see Section 2.4).

Unfortunately, SSS does not tolerate malicious dealers who distribute invalid shares, nor malicious players who might send invalid shares during reconstruction. To deal with this, *Verifiable Secret Sharing (VSS)* protocols enable players to verify shares from a potentially-

Algorithm 7 eVSS: A synchronous (t, n) VSS

Sharing Phase

Dealing round:

1. The dealer picks $\phi \in_R \mathbb{F}_p[X]$ of degree $t-1$ with $s = \phi(0)$, computes all shares $s_i = \phi(i)$, and commits to ϕ as $c = g^{\phi(\tau)}$.
2. Computes KZG proofs $\pi_i = g^{q_i(\tau)}$, $q_i(x) = \frac{\phi(x)-\phi(i)}{x-i}$, $\forall i \in [n]$.
3. *Broadcasts* c to all players. Then, sends (s_i, π_i) to each player $i \in [n]$ over an *authenticated, private* channel.

Verification round:

1. Each player $i \in [n]$ verifies π_i against c by checking if $e(c/g^{s_i}, g) = e(\pi_i, g^{\tau - i})$. If this check fails (or i received nothing from dealer), then i broadcasts a *complaint* against the dealer.

Complaint round:

1. If the size of the set S of complaining players is $\geq t$, the dealer is *disqualified*. Otherwise, the dealer reveals the correct shares with proofs by broadcasting $\{s_i, \pi_i\}_{i \in S}$.
2. If any one proof does not verify (or dealer did not broadcast), the dealer is disqualified. Otherwise, each $i \in [n]$ now has his correct share s_i.

Reconstruction Phase

Given commitment c and shares $(i, s_i, \pi_i)_{i \in T}$, $|T| \geq t$, the *reconstructor*:

1. Verifies each s_i, identifying a subset V of t players with valid shares.
2. Interpolates $s = \sum_{i \in V} \mathcal{L}_i^V(0) s_i = \phi(0)$.

malicious dealer [61, 89, 128, 173]. Furthermore, VSS also enables the reconstructor to verify the shares before interpolating the (wrong) secret.

Loosely speaking, VSS protocols must offer two properties against any adversary who compromises the dealer and $< t$ players: *secrecy* and *correctness*. Secrecy guarantees that no adversary learns the secret s when the dealer is honest, since a malicious one can simply reveal s. Correctness guarantees that, after the sharing phase, either any set of $\geq t$ honest players can always reconstruct s or the dealer is *disqualified*. We refer the reader to [128] for more formal VSS definitions.

8.4.1 Kate et al.'s eVSS

At a high-level, eVSS follows the style of previous VSS protocols [89, 173]. In the *dealing round*, the dealer commits to ϕ and sends each player their share and *proof* that their share is correct. In the *verification round*, each player verifies the proof for his share and, if incorrect, broadcasts a *complaint*. Finally, in the *complaint round*, the dealer resolves complaints (if any) by broadcasting the correct share of each complaining player. We give a detailed description of eVSS in Algorithm 7 and its asymptotic complexity in Table 7.1.

From Algorithm 7, eVSS's *overall communication* complexity is $\Theta(n)$ (since at most $2n + (t-1)$ shares and proofs are sent while dealing, complaining and reconstructing). eVSS's reconstruction phase is $O(t \log^2 t + n)$ time, since at most n shares have to be verified before

Algorithm 8 eJF-DKG's Sharing Phase

Dealing round: Each player i:

1. Picks $f_i \in_R \mathbb{F}_p[X]$ of degree $t-1$, sets $z_i = f_i(0)$ and $c_i = g^{f_i(\tau)}$.
2. Computes $g^{z_i} = g^{f_i(0)}$, a KZG proof $\pi_{i,0}$ for $f_i(0)$ and a NIZKPoK π_i^{DLog} for $g^{f_i(0)}$ and *broadcasts* $(c_i, g^{z_i}, \pi_{i,0}, \pi_i^{\mathsf{DLog}})$.
3. Computes shares $s_{i,j} = f_i(j)$ and KZG proofs $\pi_{i,j}$ and sends $(s_{i,j}, \pi_{i,j})$ to each $j \in [n]$ over an *authenticated, private* channel.

Verification round: For each $(c_i, g^{z_i}, \pi_{i,0}, \pi_i^{\mathsf{DLog}}, s_{i,j}, \pi_{i,j})$ from i, each j:

1. Verifies $\pi_{i,0}$ by checking $e(c_i/g^{z_i}, g) = e(\pi_{i,0}, g^{\tau - 0})$ and verifies the π_i^{DLog} NIZKPoK.
2. Verifies its share $s_{i,j}$ using $e(c_i/g^{s_{i,j}}, g) = e(\pi_{i,j}, g^{\tau - j})$.
3. If any of these checks fail (or nothing was received from i), then j broadcasts a complaint against i.

Complaint round:

1. Let S_i be the set of players complaining against i. If $|S_i| \geq t$, then i is marked as *disqualified* by all honest players. Otherwise, i broadcasts $\{s_{i,j}, \pi_{i,j}\}_{j \in S_i}$.
2. If any one proof does not verify (or i did not broadcast), then i is disqualified. Otherwise, each $j \in S_i$ now has his correct share $s_{i,j}$.
3. Let Q denote the set of players that were *not* disqualified. The agreed-upon (unknown) secret key $s = \sum_{j \in Q} z_j$. Each i sets $c = \prod_{j \in Q} c_j$, sets the *public key* $g^s = \prod_{j \in Q} g^{z_j}$, sets his share $s_i = \sum_{j \in Q} s_{j,i}$, and sets his KZG proof $\pi_i = \prod_{j \in Q} \pi_{j,i}$.

the secret can be interpolated fast in $\Theta(t \log^2 t)$ time [207]. eVSS's dealing round is $\Theta(nt)$ time, since n KZG proofs must be computed. The verification round is $\Theta(1)$ time (per player). If S is the set of complaining players, the complaint round takes $\Theta(|S|)$ time and communication for the dealer to send $|S|$ shares with proofs and $\Theta(|S|)$ time for each player to verify them. eVSS's *end-to-end time*, consisting of the sharing phase and reconstruction phase time, is $\Theta(nt)$.

8.5 Distributed Key Generation (DKG)

TSS protocols pose a key generation problem: if one party P splits s to the n signers (via SSS), P would know s and could sign on behalf of the group. This would make the TSS insecure and thus motivates *distributed key generation* (DKG) protocols [99, 172]. A (t, n) DKG protocol for discrete log cryptosystems allows n players to jointly generate a secret key $s \in_R \mathbb{F}_p$ with public key $g^s \in \mathbb{G}$ such that *only* subsets of size $\geq t$ can reconstruct s.

Unlike VSS protocols, where the dealer knows s (see Section 8.4), DKG protocols guarantee nobody learns s during the execution of the protocol. Typically, DKG protocols achieve this by having each player i secret-share its own secret z_i and setting the *final secret* s to be $\sum_i z_i$. Similar to VSS, DKG protocols must offer two security properties against any adversary who compromises $< t$ players: *secrecy* and *correctness*. Informally, secrecy guarantees that no adversary can learn any information about s beyond what is leaked by g^s. Correctness guarantees that all honest players agree on g^s and any subset with $\geq t$ honest players can reconstruct s.

8.5.1 Kate's eJF-DKG

In this thesis, we focus on improving eJF-DKG which, at a high-level, consists of n parallel executions of eVSS. We describe only its sharing phase in Algorithm 8, since it has the same reconstruction phase as eVSS. Note that eJF-DKG makes use of *non-interactive zero-knowledge proofs of knowledge (NIZKPoKs)* [48]. Although eJF-DKG is *biasable* and produces an s that is not guaranteed to be uniform, computing discrete logs on g^s is still hard [98, 126]. Also, debiasing DKG protocols is possible [99, 159, 185].

8.6 Vector Commitments (VCs)

A *vector commitment* (VC) scheme allows a *prover* P to compute a small *commitment* c of a *vector of messages* $(m_i)_{i\in[n]}$ where $m_i \in \mathbb{F}_p$. The prover P can *open* the commitment: P can reveal a message at a specific *position* in the vector to a *verifier* V who has the commitment c. To ensure P is not equivocating about the message at that position, V checks a *proof* against c. VCs also support efficiently updating proofs and the commitment after a vector update. We formalize VCs in Section 8.6.1.

VCs were first formalized by Catalano et al. [54], although the notion of committing to multiple messages appears earlier in the literature [55, 128, 142]. Kohlweiss and Rial [133] extend VCs with zero-knowledge protocols for proving correct computation of a new commitment, for opening messages at secret positions, and for proving secret updates of messages at secret position. Camenisch et al. [47] build VCs from KZG commitments (see Section 2.5.2) and Lagrange polynomials (see Section 2.4). Lai and Malavolta [134] formalize *subvector commitments (SVCs)* which support opening not just a single message m_i but any subset of messages $(m_i)_{i\in I}$ with positions from a set I.

Chepurnoy et al. [58] instantiate VCs using multivariate polynomial commitments [167]. Boneh et al. [31] instantiate VCs using hidden-order groups. Their construction has the advantage of having constant-sized public parameters. They also introduce *key-value map commitments*, which support a larger set of positions such as $[0, 2^{2\lambda}]$ rather than just $[1, n]$, where λ is a security parameter. Libert et al. [143] generalize VCs to *functional commitments (FCs)* which, instead of revealing the full message m_i when opening, can reveal $f_{\vec{x}}(\vec{v})$ where $\vec{v} = (m_i)_{i\in[n]}$ is the committed vector and $f_{\vec{x}} : \mathbb{F}_p^n \to \mathbb{F}_p$ is any linear function of the form $f_{\vec{x}}(\vec{v}) = \sum_{i\in[n]} x_i m_i$. Lai and Malavolta [134] generalize FCs to *linear map commitments (LMCs)* where the function can be any linear map $f : \mathbb{F}_p^n \to \mathbb{F}_p^q$ for some q.

8.6.1 Definitions

8.6.1.1 VC API

We define vector commitment schemes below. Our definition is almost identical to Catalano and Fiore's [54].

$\mathsf{VC.Setup}(1^\lambda, \ell) \to \mathsf{pp}, \mathsf{VK}$. Randomized algorithm that computes and returns public parameters pp and a verification key VK given security parameter λ and the maximum size ℓ of the vector.

$\mathsf{VC.Commit}(\mathsf{pp}, (m_j)_{j\in[\ell]}) \to c, \mathsf{aux}$. Deterministic algorithm that returns a vector commitment c to the sequence $(m_j)_{j\in[\ell]}$ of messages along with some *auxiliary information* aux (typically consisting of the messages themselves).

$\mathsf{VC.Open}(\mathsf{pp}, m_i, i, \mathsf{aux}) \to \pi_i$. Deterministic algorithm that returns a proof π_i that m_i is the ith message in the vector.

$\mathsf{VC.Verify}(\mathsf{VK}, c, m, i, \pi_i) \to T/F$. Deterministic algorithm that verifies the proof π_i that c commits to some sequence $m_1, \ldots, m_\ell$ where $m_i = m$.

$\mathsf{VC.Update}(\mathsf{pp}, c, m, m', j) \to c'$. Deterministic algorithm that returns a new commitment c' to the vector committed in c, except its jth message is changed from m to m'.

$\mathsf{VC.ProofUpdate}(\mathsf{pp}, c, \pi_i, m, m', j) \to \pi'_i$. Suppose the jth message m in c was updated to m' using $c' = \mathsf{VC.Update}(\mathsf{pp}, c, m, m', j)$. Given the old proof π_i for the ith message in c, this deterministic algorithm updates it to a new proof π'_i that verifies against c'. Note that i can be equal to j.

8.6.1.2 Correctness and Security

Definition 8.6.1 (Vector Commitment (VC) Scheme)**.** ($\mathsf{VC.Setup}$, $\mathsf{VC.Commit}$, $\mathsf{VC.Open}$, $\mathsf{VC.Verify}$, $\mathsf{VC.Update}$, $\mathsf{VC.ProofUpdate}$) is a secure *vector commitment (VC) scheme* if $\forall$ upper-bounds $\ell = \mathsf{poly}(\lambda)$ it satisfies the following properties:

Definition 8.6.2 (Opening Correctness)**.** $\forall$ vectors $(m_j)_{j\in[\ell]}$, $\forall$ positions $i \in [\ell]$:

$$\Pr\left[\begin{array}{c} \mathsf{pp}, \mathsf{VK} \leftarrow \mathsf{VC.Setup}(1^\lambda, \ell), \\ c, \mathsf{aux} \leftarrow \mathsf{VC.Commit}(\mathsf{pp}, (m_j)_{j\in[\ell]}), \\ \pi_i \leftarrow \mathsf{VC.Open}(\mathsf{pp}, m_i, i, \mathsf{aux}) : \\ \mathsf{VC.Verify}(\mathsf{VK}, c, m_i, i, \pi_i) = T \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 8.6.3 (Digest Update Correctness)**.** $\forall$ vectors $\vec{v} = (m_j)_{j\in[\ell]}$, $\forall$ positions $i \in [\ell], k \in [\ell]$, $\forall$ messages m'_k, let $\vec{u}$ be the same vector as $\vec{v}$ except with m'_k rather than m_k at position k:

$$\Pr\left[\begin{array}{c} \mathsf{pp}, \mathsf{VK} \leftarrow \mathsf{VC.Setup}(1^\lambda, \ell), \\ c, \mathsf{aux} \leftarrow \mathsf{VC.Commit}(\mathsf{pp}, \vec{v}), \\ \hat{c} \leftarrow \mathsf{VC.Update}(\mathsf{pp}, c, m_k, m'_k, k), \\ c', \mathsf{aux}' \leftarrow \mathsf{VC.Commit}(\mathsf{pp}, \vec{u}) : \\ c' = \hat{c} \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 8.6.4 (Proof Update Correctness)**.** $\forall$ vectors $(m_j)_{j\in[\ell]}$, $\forall$ positions $i \in [\ell], k \in [\ell]$, $\forall$ messages m'_k:

$$\Pr\left[\begin{array}{c} \mathsf{pp}, \mathsf{VK} \leftarrow \mathsf{VC.Setup}(1^\lambda, \ell), \\ c, \mathsf{aux} \leftarrow \mathsf{VC.Commit}(\mathsf{pp}, (m_j)_{j\in[\ell]}), \\ \pi_i \leftarrow \mathsf{VC.Open}(\mathsf{pp}, m_i, i, \mathsf{aux}), \\ c', \pi'_i \leftarrow \mathsf{VC.ProofUpdate}(\mathsf{pp}, c, \pi_i, m_k, m'_k, k) : \\ \mathsf{VC.Verify}(\mathsf{VK}, c', m'_k, k, \pi'_i) = T \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

Definition 8.6.5 (Position Binding Security)**.** $\forall$ adversaries $\mathcal{A}$ running in time $\mathsf{poly}(\lambda)$:

$$\Pr\left[\begin{array}{c} \mathsf{pp}, \mathsf{VK} \leftarrow \mathsf{VC.Setup}(1^\lambda, \ell), \\ (c, m, m', i, \pi, \pi') \leftarrow \mathcal{A}(1^\lambda, \mathsf{pp}) : \\ \mathsf{VC.Verify}(\mathsf{VK}, c, m, i, \pi) = T \wedge \\ \mathsf{VC.Verify}(\mathsf{VK}, c, m', i, \pi') = T \wedge \\ m \neq m' \end{array}\right] \leq \mathsf{negl}(\lambda)$$

Chapter 9

Constant-sized, Univariate Polynomial Commitments with Faster Proofs

In this section, we improve KZG's $\Theta(nt)$ time for computing n proofs for a degree-bound t polynomial to $\Theta(n \log t)$ time.

9.1 Authenticated Polynomial Multipoint Evaluation Trees (AMTs)

Our key technique is to commit to the quotients in a polynomial multipoint evaluation (see Section 2.3), obtaining an *authenticated multipoint evaluation tree* (AMT). However, our new AMT evaluation proofs are logarithmic-sized, whereas KZG proofs are constant-sized. Throughout this section, we restrict ourselves to computing AMTs at points $\{1, 2, \ldots, n\}$ on polynomials of degree $t - 1 < n$. (In Section 11.2.2.1, we discuss generalizing to any set of points.) Finally, in Section 9.1.7, we show AMT evaluation proofs are secure under q-SBDH. In contrast, KZG proofs are secure under a weaker assumption called q-SDH [29].

9.1.1 Computing n Evaluations Proofs in $n \log^2 n$ time

KZG evaluation proofs leverage the *polynomial remainder theorem*: $\forall i \in \mathbb{F}_p, \exists q_i$ of degree $t - 1$ such that $\phi(x) = q_i(x)(x - i) + \phi(i)$. Specifically, a constant-sized KZG proof for $\phi(i)$ is just a commitment to the quotient polynomial q_i (see Section 2.5) and takes $\Theta(t)$ time to compute. Thus, computing KZG proofs for each $i \in [n]$ takes $\Theta(nt)$ time. We improve on this by looking at $\phi(x)$ from the lens of a polynomial multipoint evaluation [207].

For example, consider the multipoint evaluation of ϕ at all $i \in [8]$ from Figure 2-1. Note that every node in the multipoint evaluation tree stores a quotient and a remainder obtained by dividing the parent node's remainder by its accumulator polynomial (see Section 2.3). The first key idea is that, for any evaluated point $i \in [8]$, $\phi(x)$ can be expressed as $\phi(i)$ plus a linear combination of quotients and accumulator polynomials along the path to $\phi(i)$'s leaf in the multipoint evaluation tree.

For example, consider $i = 1$, which has the left-most path in tree. Start with the root

node in Figure 2-1, which says:

$$\phi(x) = q_{1,8}(x)(x-1)\dots(x-8) + r_{1,8}(x).$$

Then, expand $r_{1,8}(x)$ by going left in the tree (down towards $\phi(1)$'s leaf), obtaining:

$$\begin{aligned}\phi(x) = {} & q_{1,8}(x)(x-1)(x-2)(x-3)(x-4)\cdots(x-8)\\ & + q_{1,4}(x)(x-1)(x-2)(x-3)(x-4) + r_{1,4}.\end{aligned}$$

Repeat this process recursively by replacing $r_{1,4}(x)$ and then $r_{1,2}(x)$ to get:

$$\begin{aligned}\phi(x) = {} & q_{1,8}(x)(x-1)(x-2)(x-3)(x-4)\cdots(x-8)\\ & + q_{1,4}(x)(x-1)(x-2)(x-3)(x-4)\\ & + q_{1,2}(x)(x-1)(x-2)\\ & + q_{1,1}(x)(x-1) + \phi(1).\end{aligned}$$

Note that $\phi(x)$ can be re-expressed similarly for any other points $i \in [2, n]$. Importantly, note that there are only $\Theta(n)$ quotient and accumulator polynomials shared by all $\phi(i)$ expressions, $i \in [n]$.

Our second key idea follows naturally: we commit to all these $\Theta(n)$ quotient polynomials in the multipoint evaluation of ϕ. This gives us logarithmic-sized evaluation proofs for any point $i \in [n]$. We call these proofs *AMT proofs*. For example, in Figure 2-1, the AMT proof for $\phi(4)$ would be $\{g^{q_{1,8}(\tau)}, g^{q_{1,4}(\tau)}, g^{q_{3,4}(\tau)}, g^{q_{4,4}(\tau)}\}$, where τ denotes the trapdoor used in KZG commitments (see Section 2.5).

Recall that a multipoint evaluation of ϕ at n points takes $\Theta(n \log^2 n)$ time (see Section 2.3). This asymptotically dominates the $\Theta(n \log n)$ time to commit to the quotients. Thus, *for now*, the time to compute an AMT is $\Theta(n \log^2 n)$. We explain how to speed this up to $\Theta(n \log t)$ in Section 9.1.4 for a carefully selected set of n evaluation points.

9.1.2 Verifying our New Evaluation Proofs

The next question is how to verify our new logarithmic-sized AMT proofs. Recall that, given any point i, $\phi(x)$ can be expressed as:

$$\phi(x) = \phi(i) + \sum_{w \in \mathsf{path(i)}} q_w(x) a_w(x) \tag{9.1}$$

where $\mathsf{path}(i)$ is the set of nodes along the path from the root to $\phi(i)$ and q_w and a_w denote the quotient and accumulator polynomials stored at node w in the multipoint evaluation tree (see Figure 2-1). How can we verify a proof $\pi_i = \left(g^{q_w(\tau)}\right)_{w \in \mathsf{path}(i)}$ for $\phi(i) = y_i$? We simply

use a bilinear map to check that Equation (9.1) holds at $x = \tau$:

$$
\begin{aligned}
e(g^{\phi(\tau)}, g) &\stackrel{?}{=} e(g^{y_i}, g) \prod_{w \in \mathsf{path}(i)} e(g^{q_w(\tau)}, g^{a_w(\tau)}) \Leftrightarrow \qquad (9.2)\\
e(g,g)^{\phi(\tau)} &\stackrel{?}{=} e(g,g)^{y_i} \prod_{w \in \mathsf{path}(i)} e(g,g)^{q_w(\tau) a_w(\tau)} \Leftrightarrow \\
e(g,g)^{\phi(\tau)} &\stackrel{?}{=} e(g,g)^{y_i + \sum_{i \in \mathsf{path}(w)} q_w(\tau) a_w(\tau)} \Leftrightarrow \\
\phi(\tau) &\stackrel{?}{=} y_i + \sum_{w \in \mathsf{path}(i)} q_w(\tau) a_w(\tau).
\end{aligned}
$$

This is reminiscent of how KZG proofs are verified by checking that $\phi(x) = q_i(x)(x - i) + \phi(i)$ holds at $x = \tau$ (see Section 2.5). However, note that *the verifier needs to have the $g^{a_w(\tau)}$ accumulator commitments*, which are not part of the AMT proof. This implies AMT verifiers must have $O(n)$ public parameters, whereas KZG verifiers only need $\{g^\tau\}$ as their public parameters (see Section 2.5). Fortunately, in Section 9.1.6 we reduce the verifers' public parameters to just $\Theta(\log t)$.

9.1.3 Homomorphic Proofs

AMT proofs are *homomorphic*: a proof for $f_1(j)$ and a proof for $f_2(j)$, can be *aggregated* into a proof for $(f_1 + f_2)(j)$. The intuition for why this holds is that "adding up" the multipoint evaluation trees of two polynomials ϕ and ρ at the same points (i.e., at $X = \{\omega_N^{j-1}\}_{j \in [n]}$) results in a multipoint evaluation tree of their sum $\phi + \rho$ (also at X). In more detail, let $q_{w,[\psi]}$ denote the quotient polynomial at node w in ψ's multipoint evaluation tree (at X). Then, one can show that $q_{w,[\phi+\rho]} = q_{w,[\phi]} + q_{w,[\rho]}$ and that $g^{q_{w,[\phi+\rho]}(\tau)} = g^{q_{w,[\phi]}(\tau) + q_{w,[\rho]}(\tau)} = g^{q_{w,[\phi]}(\tau)} g^{q_{w,[\rho]}(\tau)}$. In other words, given an AMT for ϕ and an AMT for ρ, we can obtain an AMT for $\phi + \rho$ by multiplying quotient commitments at each node. It follows that a proof for $f_1(a)$ and one for $f_2(a)$ can be aggregated into a proof for $(f_1 + f_2)(a)$ by multiplying commitments at each node.

9.1.4 Better AMTs via Roots of Unity

Instead of evaluating ϕ at points $\{1, 2, 3, \ldots, n\}$, we assume $n = 2^m$ and evaluate ϕ at all n nth roots of unity in $\mathbb{F}_p$. Specifically, we compute $\phi(\omega_n^{i-1})$ rather than $\phi(i)$, where ω_n is a primitive nth root of unity. (We can generalize to any n by using the first n Nth roots of unity, where $N = 2^m$ is the smallest value such that $N \geq n$.)

The main benefit of using roots of unity is they give rise to simpler accumulator polynomials of the form $(x^{2^k} + c)$ in the multipoint evaluation tree (for some c). This speeds up the multipoint evaluation (see Section 9.1.5), since dividing degree-bound $2n$ polynomials by $(x^n + c)$ can be done in $\Theta(n)$ rather than $\Theta(n \log n)$ time. In Section 9.1.5, we show this new, optimized AMT proof is $\lfloor \log(t-1) \rfloor + 1$ group elements and computing an AMT takes $\Theta(n \log t)$ time.

The $(x^{2^k} + c)$ form of the accumulators is best illustrated with an example. Let $n = 8$ and ω_8 denote a primitive 8th root of unity. Previously, in Figure 2-1, the evaluation points

$\{1, 2, \ldots, 8\}$ were ordered as $\langle (x-1), (x-2), \ldots, (x-8)\rangle$ monomials in the leaves. Then, the accumulators were computed by multiplying "up the tree," culminating in the root accumulator $\prod_{i\in[8]}(x-i)$. In our case, the evaluation points are $\{\omega_8^{i-1}\}_{i\in[8]}$ but we reorder them using a bit-reversal permutation [210] as $\langle (x-\omega_8^0), (x-\omega_8^4), (x-\omega_8^2), (x-\omega_8^6), (x-\omega_8^1), (x-\omega_8^5), (x-\omega_8^3), (x-\omega_8^7)\rangle$. This ordering ensures that, as we multiply "up the tree," all accumulators are of the form $(x^{2^k}+\omega_8^j)$ for some j.

Let us see exactly how this happens. The parent accumulator of the first two leaves $(x-\omega_8^0)$ and $(x-\omega_8^4)$ is their product $(x-\omega_8^0)(x-\omega_8^4) = x^2-\omega_8^4 x-\omega_8^0 x+\omega_8^0\omega_8^4$. Since $\omega_n^i\omega_n^j = \omega_n^{(i+j) \bmod n}$ [63], this equals $x^2-x(\omega_8^4+\omega_8^0)+\omega_8^4$. Since $\omega_n^{k+n/2} = -\omega_n^k$ [63], this equals $(x^2+\omega_8^4)$. The remaining accumulators after $(x^2+\omega_8^4)$ on this level are $\{(x^2+\omega_8^0), (x^2+\omega_8^6), (x^2+\omega_8^2)\}$. Recursing on the next level, its accumulators are $\langle (x^4+\omega_8^4), (x^4+\omega_8^0)\rangle$. Finally, the root will be $(x^8-\omega_8^0) = (x^8-1) = \prod_{i=0}^{7}(x-\omega_8^i)$.

9.1.5 Prover Time and Proof Sizes

We will restrict ourselves to our $n = 2^m$ and $\deg\phi = t-1 < n$ setting. We first show that computing our optimized, roots-of-unity-based AMT takes $O(n\log t)$ time (see Section 9.1.4). The key observation is that, when computing the AMT, divisions at higher levels (i.e., closer to the root) in the tree are *trivial* and need not be performed. Specifically, at sufficiently high levels, the degree of the divisors (i.e., accumulators) are larger than the degrees of the dividends (i.e., remainders), and always give quotients equal to zero. Since zero quotients can be easily recreated by verifiers, their commitments need not be included in the proof. We expand on this next.

Let us number levels differently, from $\log n$ (the root) to 0 (the leaves), so that level i has $n/2^i$ nodes, each with an accumulator of degree 2^i. Now, let k be the smallest value such that $2^k \le \deg\phi < 2^{k+1}$. In other words, k is the level at which accumulator degrees are $\le \deg\phi$ and thus divisions are non-trivial. Put differently, each node on level k will be the root node of an (authenticated) multipoint evaluation (sub)tree. We argue that the time to compute any one such subtree is $O(2^k\log 2^k)$ and, since there are $n/2^k$ such subtrees, the final AMT takes $O(n\log 2^k) = O(n\log t)$ time since $2^k \le t-1 = \deg\phi$. We prove this inductively next.

At the root node of a level k subtree, the dividend $d_k = \phi$ has $\deg d_k < 2^{k+1}$ (by definition of k above). The accumulator a_k has $\deg a_k = 2^k$. Thus, the quotient $q_k = d_k/a_k$ will have $\deg q_k = \deg d_k - \deg a_k < 2^{k+1} - 2^k = 2^k$ and the remainder $r_k = d_k \bmod a_k$ will have $\deg r_k < \deg a_k = 2^k$. The division at this level will only take $O(\deg d_k) = O(2^{k+1})$ time, thanks to the $(x^{2^k}+c)$ form of a_k. Committing to the quotient will take $O(2^k)$ time. To summarize, at level k we are doing $O(2^{k+1})$ work and $\deg d_k < 2^{k+1}, \deg a_k = 2^k, \deg q_k < 2^k, \deg r_k < 2^k$.

Next, we argue that the amount of work *per node* on level $k-1$ is half the work per node at level k. This is because (1) the dividend d_{k-1} is set to the remainder r_k from the parent, so $\deg d_{k-1} < 2^k$, (2) $\deg a_{k-1} = 2^{k-1}$, (3) $\deg q_{k-1} = \deg d_{k-1} - \deg a_{k-1} < 2^k - 2^{k-1} = 2^{k-1}$ and (4) $\deg r_{k-1} < \deg a_{k-1} = 2^{k-1}$. Thus, at level $k-1$, the division takes $O(2^k)$ time and committing to the quotient takes $O(2^{k-1})$ time. As a result, the time to compute the subtree can be expressed as $T(2^{k+1}) = 2T(2^{k+1}/2) + O(2^{k+1}) = O(2^k\log 2^k)$.

Finally, an AMT proof is $O(\log t)$-sized. Recall that quotients in the AMT are non-zero only at levels k and below, where $2^k \le t-1 < 2^{k+1}$. Thus, an AMT proof will only have

non-zero quotients at levels $k, k-1, k-2, \ldots, 1, 0$. Since $k = \lfloor \log_2(t-1) \rfloor$ the exact proof size is $\lfloor \log_2(t-1) \rfloor + 1$ group elements.

9.1.6 Keeping (Almost) the Same Public Parameters

Do AMTs need extra public parameters, which would impose unwanted overhead in the trusted setup phase (see Section 2.6)? Recall that in KZG, given $(t-1)$-SDH public parameters, one can commit to any degree-bound t polynomials and compute *any number* of KZG evaluation proofs. In contrast, computing an AMT at $n > t-1$ points seems to require committing to degree $n > t-1$ accumulator polynomials (e.g., to the root accumulator $(x^n - 1)$). Yet this is not possible given only $(t-1)$-SDH parameters, as ensured by the $(t-1)$-polyDH assumption (see Definition 8.2.1).

Fortunately, when computing an AMT, divisions by accumulators of degree $> t-1$ always give quotient zero (see Section 9.1.5). This means that, when pairing such quotients with their accumulators during proof verification, the result will always be $1_{\mathbb{G}_T}$ (see Equation (9.2)). In other words, such pairings need never be computed and so their corresponding accumulators (of degree $> t-1$) need never be committed to. Furthermore, quotients are not problematic since they always have degree $< \deg \phi = t-1$ (or are equal to zero).

Second, AMT verifiers only need a logarithmic number of $g^{\tau^{2^k}}$ powers to recreate any accumulator commitment $g^{a_w(\tau)}$. (This is a bit worse than KZG verifiers, who only need g^{τ}.) Specifically, given a subset $\{g^{\tau^{2^k}} \mid 0 \leq k \leq \lfloor \log(t-1) \rfloor\}$ of the $(t-1)$-SDH parameters, the verifier can commit to any degree-bound t accumulator of the $(x^{2^k} + c)$ form. Thus, we impose no additional overhead in the trusted setup. In contrast, if we evaluated ϕ at $\{1, 2, \ldots, n\}$, verifiers would need all $(t-1)$-SDH public parameters to reconstruct the accumulators.

9.1.7 Security Proofs

We show our modified KZG scheme with AMT proofs satisfies *computational hiding* (see Definition 2.5.6) under the discrete log (DL) assumption and *evaluation binding* (see Definition 2.5.5) under the ℓ-Strong Bilinear Diffie-Hellman (ℓ-SBDH) assumption. These properties were originally defined in [128]. We prove these properties hold for a more general scheme that builds AMTs for an arbitrary set X of n points (rather than just for the set of roots of unity). For this scheme, $\mathsf{Poly.Setup}$ returns not only ℓ-SDH public parameters, but also the accumulator commitments necessary to verify AMT proofs. In other words, given an evaluation point $x^* \in X$, verifiers have access to accumulators $\{g^{a_w(\tau)}\}_{w \in \mathsf{path}(x^*)}$ necessary to verify x^*'s AMT proof. (Recall that accumulators of degree $> \deg(\phi)$ can be discarded; see Section 9.1.6.)

Computational Hiding Proof. Suppose there exists an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ that breaks computational hiding for polynomials of degree d (see Definition 2.5.6) and outputs $\phi(\hat{x})$ for an unqueried $\hat{x} \neq x_i, \forall i \in [d]$ where $d = \deg \phi$. Then, we show how to build an adversary $\mathcal{B}$ that takes as input a random discrete log instance (g, g^a) and uses $\mathcal{A}$ to break it and output a. (Our proof is in the same style as Kate et al.'s proof for $\mathsf{PolyCommit}_{\mathsf{DL}}$'s computational hiding [129].)

$\mathcal{B}$ runs $\mathsf{Poly.Setup}(1^\lambda, d)$ which picks $\tau \in_R \mathbb{F}_p$ and outputs public parameters $\mathsf{pp} = \mathsf{PP}_d^{\mathsf{SDH}}(g;\tau)$. Importantly, since $\mathcal{B}$ runs the setup, $\mathcal{B}$ will know the trapdoor τ. $\mathcal{B}$ calls $\mathcal{A}_0$, which picks d distinct points $x_i \in X, \forall i \in [d]$. Then, $\mathcal{B}$ picks d random evaluations $y_i \in \mathbb{F}_p$, one for each x_i. $\mathcal{B}$ also picks a random $x_0 \in X$ different than all x_i's. For notational convenience, let $y_0 = a$. (Although $\mathcal{B}$ does not know a, it will be sufficient for him to know $g^{y_0} = g^a$.) Note that $(x_i, y_i)_{i=0}^d$ determines a degree d polynomial ϕ where $\phi(x_i) = y_i, \forall i \in [0, d]$. Since $\mathcal{B}$ does not know a (only g^a), it will interpolate ϕ's commitment $g^{\phi(\tau)}$ "in the exponent" as:

$$g^{\phi(\tau)} = \prod_{i \in [0,d]} (g^{y_i})^{\mathcal{L}_i^T(\tau)}.$$

Here, $T = \{x_i\}_{i\in[0,d]}$ and recall that $\mathcal{L}_i^T(\tau) = \prod_{j\in T, j\neq i}(\tau - x_j)/(x_i - x_j)$ (see Section 2.4). To summarize, $\mathcal{B}$ "embeds" the (g, g^a) challenge in an (unknown-to-$\mathcal{B}$-but-determined) polynomial ϕ with commitment $c = g^{\phi(\tau)}$.

Next, $\mathcal{B}$ has to simulate AMT proofs π_i for $y_i = \phi(x_i), \forall i \in [d]$. To do this, recall that at each node w in the AMT, we have quotient and remainder polynomials q_w, r_w such that $r_{\mathsf{parent}(w)} = q_w a_w + r_w$ (see Figure 2-1). Also, recall that $\mathcal{B}$ knows τ so he can compute accumulator evaluations $a_w(\tau), \forall$ nodes w in the AMT. Now, $\mathcal{B}$ can simulate proofs as follows.

For the root node $w = \varepsilon$, we have $r_{\mathsf{parent}(\varepsilon)} = \phi$, so $\mathcal{B}$ picks a random $r_\varepsilon(\tau) \in \mathbb{F}_p$, and computes the root quotient commitment as $g^{q_\varepsilon(\tau)} = (g^{\phi(\tau)}/g^{r_\varepsilon(\tau)})^{\frac{1}{a_\varepsilon(\tau)}}$. At the next level, consider the children nodes u and v of the root ε. For each child $w \in \{u, v\}$, $\mathcal{B}$ must commit to a quotient q_w that satisfies $r_\varepsilon(\tau) = q_w(\tau)a_w(\tau) + r_w(\tau)$ for some r_w. So $\mathcal{B}$ proceeds similarly: for each child $w \in \{u, v\}$, he picks a random $r_w(\tau)$ and computes a commitment $g^{q_w(\tau)} = (g^{r_\varepsilon(\tau)}/g^{r_w(\tau)})^{\frac{1}{a_w(\tau)}}$. $\mathcal{B}$ will do this recursively until it reaches leaf nodes in the AMT. For each leaf l, instead of picking $r_l(\tau)$ randomly, $\mathcal{B}$ will set it to the y_i corresponding to that leaf. This way, $\mathcal{B}$ can simulate quotient commitments $\{g^{q_w(\tau)}\}_{w\in\mathsf{path}(x_i)}$ for all $i \in [d]$ that pass the AMT proof verification in Equation (9.2).

Next, $\mathcal{B}$ calls $\mathcal{A}$ with $(\mathsf{pp}, c, (x_i, y_i, \pi_i)_{i=1}^d)$ as input, hoping that $\mathcal{A}$ outputs another point $\hat{x}$ and its evaluation $\phi(\hat{x})$. Since $\mathcal{A}$ breaks computational hiding, this happens with non-negligible probability. (Note that $\mathcal{B}$ can check $\mathcal{A}$ succeeds by interpolating $g^{\phi(\hat{x})}$ "in the exponent".) When $\mathcal{A}$ succeeds, if $\hat{x} = x_0$, then $a = \phi(\hat{x})$, so $\mathcal{B}$ breaks discrete log on (g, g^a). Otherwise, $\mathcal{B}$ uses the first d points $(x_i, y_i = \phi(x_i))_{i\in[d]}$ and this new distinct $(\hat{x}, \phi(\hat{x}))$ point to interpolate ϕ and as a result obtain $a = \phi(x_0)$. (Recall that, by Definition 2.5.6, we have $\hat{x} \neq x_i, \forall i \in [d]$.) As a result, $\mathcal{B}$ breaks discrete log on (g, g^a).

Evaluation Binding Proof. Suppose there exists an adversary $\mathcal{A}$ that outputs a commitment c, with two contradicting proofs π, π' attesting that $\phi(k)$ is equal to v and v', respectively. We show how to build another adversary $\mathcal{B}$ that breaks q-SBDH. First, $\mathcal{B}$ runs $\mathcal{A}$ to get $(c, \pi, \pi', \phi(k), v, v')$. Let $W = \mathsf{path}(k)$ denote the nodes along k's path in the AMT. Let $(\pi_i)_{i\in W}$ denote the quotient commitments in π. Similarly, let $(\pi'_i)_{i\in W}$ denote the quotient

commitments in π'. Since both proofs verify, we have:

$$e(c,g) = e(g^v, g) \prod_{i \in W} e(\pi_i, g^{a_i(\tau)})$$
$$e(c,g) = e(g^{v'}, g) \prod_{i \in W} e(\pi'_i, g^{a_i(\tau)}).$$

Dividing the first equation by the second, we get:

$$1_{\mathbb{G}_T} = \frac{e(g^v, g)}{e(g^{v'}, g)} \frac{\prod_{i \in W} e(\pi_i, g^{a_i(\tau)})}{\prod_{i \in W} e(\pi'_i, g^{a_i(\tau)})} \Leftrightarrow$$
$$1_{\mathbb{G}_T} = e(g^{v-v'}, g) \prod_{i \in W} \frac{e(\pi_i, g^{a_i(\tau)})}{e(\pi'_i, g^{a_i(\tau)})} \Leftrightarrow$$
$$e(g^{v'-v}, g) = \prod_{i \in W} e(\pi_i / \pi'_i, g^{a_i(\tau)}).$$

Now, recall that one of the accumulators $(a_i(x))_{i \in W}$ is the monomial $(x-k)$, and all the other $a_i(x)$'s contain $(x-k)$ as a term, which means it can be factored out of them. Thus, since $(x-k)$ perfectly divides all $a_i(x)$'s, let $r_i(x) = a_i(x)/(x-k), \forall i \in W$. Importantly, the adversary $\mathcal{B}$ can compute all $r_i(x)$'s in polynomial time, since it can reconstruct all the accumulator polynomials $(a_i(x))_{i \in W}$. As a result, $\mathcal{B}$ can compute all commitments $(g^{r_i(\tau)})_{i \in W}$. Then, $\mathcal{B}$ breaks ℓ-SBDH as follows:

$$e(g^{v'-v}, g) = \prod_{i \in W} e(\pi_i / \pi'_i, g^{r_i(\tau)(\tau-k)})$$
$$e(g^{v'-v}, g) = \prod_{i \in W} e(\pi_i / \pi'_i, g^{r_i(\tau)})^{(\tau-k)}$$
$$e(g^{v'-v}, g) = \left[\prod_{i \in W} e(\pi_i / \pi'_i, g^{r_i(\tau)}) \right]^{(\tau-k)}$$
$$e(g,g)^{\frac{1}{\tau-k}} = \left[\prod_{i \in W} e(\pi_i / \pi'_i, g^{r_i(\tau)}) \right]^{\frac{1}{v'-v}}.$$

9.2 Vector Commitments (VCs) from AMTs

Our AMT technique for precomputing evaluation proofs at N points in a polynomial commitment (see Chapter 9) naturally gives rise to a bounded Vector Commitment (VC) scheme similar to [54, 58]. Our scheme can be regarded as the univariate polynomial counterpart of Chepurnoy et al.'s scheme [58], which is based on multivariate polynomials.

9.2.1 Overview

Let $N = 2^k$ be the maximum size of vectors our scheme can commit to and let ω_N be a primitive Nth root of unity. We encode the message m_i at position i in the vector as $\phi(\omega_N^{i-1}) = m_i, \forall i \in [N]$ using a degree $N-1$ polynomial ϕ, similar to [47]. We then use

AMT proofs to prove that m_i is the correct message at position i. Furthermore, we use the homomorphism of KZG commitments (see Section 2.5.2.1) and of AMT proofs (see Section 9.1.3) to update VC digests and proofs, respectively. Lastly, our VC scheme can be turned into a *subvector commitment scheme (SVC)* [134] using KZG batch proofs (see Section 2.5.2.1).

9.2.2 Updating VC Proofs

Recall from Section 8.6 that a VC scheme must support updating proofs after the vector has been changed. Specifically, if the jth message in the vector committed in c changes from m to m', then given a proof π_i for any position $i \in [\ell]$, we should be able to update it to a new proof π_i' that verifies against the updated vector commitment c', which has m' rather than m at position j. We describe below how to update proofs by leveraging the AMT homomorphism (see Section 9.1.3).

Let $\phi(x)$ be the polynomial committed in c and $\phi'(x)$ be the new polynomial of the updated vector committed in c'. Then, $\phi'(x) = \phi(x) + \mathcal{L}_{j-1}(x)(m' - m)$. This implies that the AMT of ϕ' is just the AMT of ϕ homomorphically combined with the AMT of $\mathcal{L}_{j-1}(x)(m' - m)$. As a consequence, the proof π_i' is just the proof π_i homomorphically combined with the AMT of $\mathcal{L}_{j-1}(x)(m' - m)$, but only at nodes that intersect $\mathsf{path}(i)$. Thus, to update proofs, we can compute an AMT over $\mathcal{L}_{j-1}(x)(m' - m)$ and homomorphically combine it with π_i'. Note that this AMT will only consist of a single path to the evaluation $\mathcal{L}_{j-1}(\omega_N^{j-1}) = m' - m$.

In practice, to save computation time, AMTs for all $\mathcal{L}_i(x), i \in [0, \ell-1]$ can be precomputed during $\mathsf{VC.Setup}$ (see Section 8.6.1). Then, when a proof π_i must be updated after a change at position j from m to m', the precomputed AMT for $\mathcal{L}_{j-1}(x)$ can be multiplied by the $m' - m$ delta. This results in an AMT for $\mathcal{L}_{j-1}(x)(m' - m)$ which can now be used to update π_i.

9.2.3 Construction

We give a detailed description of our VC scheme below. Security of our VC scheme follows directly from our security proof in Section 9.1.7 that AMT-based evaluation proofs satisfy *Evaluation Binding* as defined in Definition 2.5.5.

$\mathsf{VC.Setup}(1^\lambda, \ell) \to \mathsf{pp}, \mathsf{VK}$. Generates bilinear pairing parameters $\langle \mathbb{G}, \mathbb{G}_T, p, g, e \rangle \leftarrow \mathcal{G}_{\mathsf{prime}}(1^\lambda)$ Generates random $(\ell-1)$-SDH public parameters $\mathsf{PP}_{\ell-1}^{\mathsf{SDH}}(g; \tau)$. Let $N = 2^k$ denote the smallest power of two such that $N \geq \ell$ and let ω_N denote a primitive Nth root of unity in $\mathbb{F}_p$. For all $i \in [0, \ell-1]$, computes KZG commitments $g^{\mathcal{L}_i(\tau)}$ to the Lagrange polynomial $\mathcal{L}_i(x) = \prod_{j \in [0,\ell-1], j \neq i} (x - \omega_N^j)/(\omega_N^i - \omega_N^j)$. (These are needed to update digests later.) Sets $\mathsf{pp} = \left(N, \omega_N, \mathsf{PP}_{\ell-1}^{\mathsf{SDH}}(g; \tau), \left(g^{\mathcal{L}_i(\tau)}\right)_{i \in [0,\ell-1]}\right)$. (Optionally, can also include AMTs for all Lagrange polynomials in pp, to make updating proofs faster.) Sets $\mathsf{VK} = \left(g, (g^{\tau^{2^i}})_{i \in [0, \lfloor \log_2(\ell-1) \rfloor]}\right)$.

$\mathsf{VC.Commit}(\mathsf{pp}, (m_j)_{j \in [\ell]}) \to c, \mathsf{aux}$. Let $\phi(x) = \sum_{j \in [\ell]} \mathcal{L}_{j-1}(x) m_j$ be the polynomial that "encodes" the vector. Note that $\phi(\omega_N^{j-1}) = m_j, \forall j \in [\ell]$. Computes an AMT over ϕ,

evaluating it at the first ℓ Nth roots of unity (see Section 9.1.1). Stores the AMT in aux. Returns a KZG commitment $c = g^{\phi(\tau)}$ and aux.

$\mathsf{VC.Open}(\mathsf{pp}, m_i, i, \mathsf{aux}) \to \pi_i$. Parses the AMT out from the auxiliary data aux and returns an AMT proof for $\phi(\omega_N^{i-1}) = m_i$.

$\mathsf{VC.Verify}(\mathsf{VK}, c, m, i, \pi_i) \to T/F$. Verifies the AMT proof π_i for $\phi(\omega_N^{i-1}) = m_i$ (see Section 9.1.2).

$\mathsf{VC.Update}(\mathsf{pp}, c, m, m', j) \to c'$. Sets $c' = c\left(g^{\mathcal{L}_{j-1}(\tau)}\right)^{m'-m}$.

$\mathsf{VC.ProofUpdate}(\mathsf{pp}, c, \pi_i, m, m', j) \to \pi_i'$. Computes an AMT over $\mathcal{L}_{j-1}(x)$ (or fetches the precomputed one from the public parameters pp, if any). Multiplies this AMT by $(m'-m)$. Homomorphically combines π_i with the AMT for $\mathcal{L}_{j-1}(x)(m'-m)$ only at the nodes they intersect at (see Section 9.2.2). The final result will be the updated proof π_i'.

Chapter 10

Scalable Threshold Cryptosystems

10.1 Scalable Threshold Signatures (TSS)

In this section, we show how to reduce the time to aggregate a (t, n) BLS threshold signature from $\Theta(t^2)$ to $\Theta(t \log^2 t)$. Although we focus on BLS, our techniques can be used in any threshold cryptosystem (not just signatures) whose secret key lies in a prime-order field $\mathbb{F}_p$. This includes ElGamal signatures [97, 114, 171], ElGamal encryption [74], Schnorr signatures [98, 196] (but not RSA-based schemes, whose secret key does not lie in a prime-order field [192]).

Recall from Section 8.3 that (t, n) threshold signature aggregation has two phases: (1) computing Lagrange coefficients and (2) exponentiating signature shares by these coefficients. Unfortunately, when implemented naively in $\Theta(t^2)$ time, the time to compute Lagrange coefficients dominates the cost of aggregation (see Figure 10-1). In fact, current descriptions and implementations of threshold schemes all seem to use this inefficient scheme, which we dub *naive Lagrange* [28, 59, 77, 154, 204].

We make three contributions. First, we adapt the *fast polynomial interpolation* from [207] to compute just the Lagrange coefficients fast in $\Theta(t \log^2 t)$ time. We call this scheme *fast Lagrange*. Second, we speed up this scheme by using roots-of-unity rather than $\{1, 2, \ldots, n\}$ as the signer IDs. Third, we implement threshold BLS signatures based on fast Lagrange and show they outperform the naive ones as early as $n = 511$ (see Section 10.4.1).

10.1.1 Fast Lagrange-based BLS

Recall that a *Lagrange polynomial* $\mathcal{L}_i^T(x)$ is defined as:

$$\mathcal{L}_i^T(x) = \prod_{\substack{j \in T \\ j \neq i}} \frac{x - j}{i - j} \tag{10.1}$$

First, let us rewrite $\mathcal{L}_i^T(x)$ as:

$$\mathcal{L}_i^T(x) = \frac{N_i(x)}{D_i} \tag{10.2}$$

$$N_i(x) = \frac{N(x)}{x - i} = \prod_{\substack{j \in T \\ j \neq i}} (x - j) \tag{10.3}$$

$$N(x) = \prod_{i \in T} (x - i) \tag{10.4}$$

$$D_i = N_i(i) = \prod_{\substack{j \in T \\ j \neq i}} (i - j) \tag{10.5}$$

Our goal is to quickly compute $\mathcal{L}_i^T(0)$ for each signer ID $i \in T$. In other words, we need to quickly compute all $N_i(0)$'s and all D_i's.

First, given the set of signer IDs T, we interpolate $N(x)$ in $\Theta(t \log^2 t)$ time by starting with the $(x - i)$'s as leaves of a tree and "multiplying up the tree." Second, we can compute all $N_i(0) = N(0)/(-i)$ in $\Theta(t)$ time. (Note that $N(0)$ is just the first coefficient of $N(x)$.) However, computing $D_i, \forall i \in T$ appears to require $\Theta(t^2)$ time. Fortunately, the *derivative* $N'(x)$ of $N(x)$ evaluated at i is exactly equal to D_i [207]. Thus, an $\Theta(t \log^2 t)$ multipoint evaluation of $N'(x)$ at all $i \in T$ can efficiently compute all D_i's!

To see why $N'(i) = D_i$, it is useful to look at the closed form formula for $N'(x)$ obtained by applying the product rule of differentiation (i.e., $(fg)' = f'g + fg'$). For example, for $N(x) = (x - 1)(x - 2)(x - 3)$:

$$\begin{aligned} N'(x) &= (x - 2)(x - 3) + (x - 1)(x - 3) + (x - 1)(x - 2) \\ &= N_1(x) + N_2(x) + N_3(x) \end{aligned}$$

In general, we can prove that $N'(x) = \sum_{i \in T} N_i(x)$ where $\deg N'(\cdot) = t - 1$. Since $N_j(i) = 0$ for all $i \neq j$ (see Equation (10.2)), it follows that $N'(i) = N_i(i) + 0 = D_i$. Lastly, computing $N'(x)$ only takes $\Theta(t)$ time via polynomial differentiation. Specifically, if $N = (c_t, c_{t-1}, \ldots, c_1, c_0)$, then $N' = (t \cdot c_t, (t - 1)c_{t-1}, \ldots, 2c_2, c_1)$.

To summarize, given a set T of signer IDs, we can compute the Lagrange coefficients $\mathcal{L}_i^T(0) = N_i(x)/N'(i)$ by (1) computing $N(x)$ in $\Theta(t \log^2 t)$ time, (2) computing all $N_i(0)$'s in $\Theta(t)$ time, (3) computing $N'(x)$ in $\Theta(t)$ time and (4) evaluating $N'(x)$ at all $i \in T$ in $\Theta(t \log^2 t)$ time. This reduces the time to compute all $\mathcal{L}_i^T(0)$'s from $\Theta(t^2)$ to $\Theta(t \log^2 t)$.

10.1.2 Further Speed-ups via Roots of Unity

The fast Lagrange technique works for any threshold cryptosytem whose secret key s lies in prime-order field $\mathbb{F}_p$. However, for fields that support roots of unity, further speed-ups are possible. (A caveat is that pairings on the underlying elliptic curve can be up to $2\times$ slower.) Without loss of generality, assume the total number of signers n is a power of two and let ω_n denote a primitive nth root of unity in $\mathbb{F}_p$. If we replace the $\{1, \ldots, n\}$ signer IDs with roots of unity $\{\omega_n^{i-1}\}_{i \in [n]}$, then $N'(x)$ can be evaluated at any subset of signer IDs with a single Fast Fourier Transform (FFT). This is much faster than a polynomial multipoint evaluation, which performs many polynomial divisions, each involving many FFTs.

Our fast Lagrange implementation from Section 10.4.1 takes advantage of this optimization. Furthermore, we use roots of unity to compute inverses faster in both our naive and fast Lagrange implementations (see Section 10.4.1). For example, in naive Lagrange, we compute $N(0) = \prod_{i \in T}(0 - \omega_n^i)$ much faster as $(-1)^{|T|} \cdot \omega_n^{\sum_{i \in T} i}$.

10.2 Scalable Verifiable Secret Sharing (VSS)

In this section, we scale (t, n) VSS protocols to large n in the difficult case when $t > n/2$. Specifically, we reduce eVSS's dealing time from $\Theta(nt)$ to $\Theta(n \log t)$ by replacing KZG proofs with AMT proofs. We call this new VSS protocol AMT VSS and describe it below.

10.2.1 Faster Dealing via AMTs

The difference between AMT VSS and eVSS is very small. First, players' shares are computed as $s_i = \phi(\omega_N^{i-1})$ (rather than $\phi(i)$ as in eVSS), where N is the smallest power of two $\geq n$. Second, instead of using (slow) KZG proofs, the dealer computes an AMT for ϕ at points $\{\omega_N^{i-1}\}_{i \in [n]}$, obtaining the shares s_i for free in the process. Then, as in eVSS, the dealer sends each player i its share s_i but now with an AMT proof π_i (see Section 9.1.1). The verification round, complaint round and reconstruction phase remain the same, except they all use AMT proofs now.

AMT VSS's dealing time is $\Theta(n \log t)$, dominated by the time to compute an AMT. This is a significant reduction from eVSS's $\Theta(nt)$ time, but comes at a small cost due to our larger AMT proofs. First, the verification round time increases from $\Theta(1)$ to $\log t$. Second, the complaint round complexity increases from $O(t)$ to $O(t \log t)$ time and communication (but we improve it in Section 10.2.2). Third, the reconstruction phase time increases from $\Theta(t \log^2 t + n)$ to $O(t \log^2 t + n \log t)$ (but we improve it in Section 10.2.3). Finally, the overall communication increases from $\Theta(n)$ to $n \log t$. Nonetheless, in Section 10.4.2, we show AMT VSS's end-to-end time is much smaller than eVSS's, which makes these increases justifiable.

10.2.2 Faster Complaints

Kate et al. previously point out that KZG batch proofs (see Section 2.5.2.1) can be used to reduce the communication and the *concrete* computational complexity of eVSS's complaint round [128]. Suppose S is the set of complaining players. Without batch proofs, the dealer only has to broadcast $|S|$ previously-computed KZG proofs and each player has to verify them by computing $2|S|$ pairings. With batch proofs, the dealer spends $\Theta(|S| \log^2 |S| + t \log t)$ time to compute the batch proof and each player spends $\Theta(|S| \log^2 |S|))$ to verify it.

While batch proofs increase asymptotic complexity for the dealer and players, the concrete complexity decreases, since players now only compute two pairings rather than $2|S|$. Furthermore, the communication complexity decreases, since only 1 proof rather than $|S|$ needs to be broadcast. Thus, AMT VSS can also use batch proofs and maintain the same performance as eVSS during the complaint round. (However, in Table 7.1, we do not assume this optimization.)

10.2.3 More Efficient Reconstruction via Memoization

In some cases, we can reduce the number of pairings computed during AMT VSS's reconstruction phase. In this phase, the reconstructor is given anywhere from t to n shares and their AMT proofs. His task is to find a subset of t valid shares and interpolate the secret. Let us first consider the *best case*, where all submitted shares are *valid*. In this case, if the reconstructor naively verifies any t AMT proofs, he spends $\Theta(t \log t)$ time. But he would be computing the same quotient-accumulator pairings multiple times (as in Equation (9.2)), since proofs with intersecting paths will share quotient commitments. By memoizing these computations, the reconstructor can verify the t proofs in $\Theta(t)$ time. Alternatively, this can be sped up by exposing a g^s public key during dealing (as in DKG protocols; see Section 10.3.2).

Now let us consider the *worst case*, where $n - t$ shares are invalid and t shares are valid. The reconstructor wants to find the t valid shares as fast as possible. Once again, he can memoize the quotient-accumulator pairings that are part of successfully validated proofs. This way, for the t valid proofs, only $\Theta(t)$ pairings need to be computed. Thus, at most $\Theta((n-t)\log t)$ pairings could possibly be computed for the invalid proofs. The worst-case reconstruction time remains $\Theta(n \log t)$ but, in practice, the number of pairings is reduced significantly by the memoization.

10.2.4 Keeping (Almost) the Same Public Parameters

The AMT VSS dealer needs $(t-1)$-SDH public parameters, just like in eVSS. This is because committing to accumulator polynomials of degree $\geq t$ is not necessary, as discussed in Section 9.1.6. In fact, adding more public parameters for committing to degree $\geq t$ polynomials would break the *correctness* of eVSS and thus of AMT VSS [128]. Specifically, if the dealer commits to a degree $\geq t$ polynomial ϕ, then different secrets could be reconstructed, depending on the subset of players whose shares are used. This is why the $(t-1)$-polyDH assumption (see Definition 8.2.1) is needed in both protocols. Finally, AMT VSS players (and the reconstructor) need $\Theta(\log t)$ public parameters to verify AMT proofs, an increase from eVSS' $\Theta(1)$ (i.e., g^τ).

10.3 Scalable Distributed Key Generation (DKG)

In this section, we scale (t, n) DKG protocols to large n in the difficult case when $t > n/2$. We start from eJF-DKG, where each player acts as an eVSS dealer (see Algorithm 8), taking $\Theta(nt)$ time to compute n KZG evaluation proofs and $\Theta(t)$ time to compute one KZG proof for $g^{f_i(0)}$ (see Algorithm 8). We simply replace eVSS with AMT VSS in eJF-DKG, obtaining a new protocol we call AMT DKG with smaller $\Theta(n \log t)$ per-player dealing time. Importantly, we keep the same KZG proof for $g^{f_i(0)}$.

Compared to eJF-DKG, AMT DKG has slightly larger communication (see Section 10.4.3.5), larger proof verification times and a slower complaint round (see Table 7.1). Fortunately, when using KZG batch proofs (see Section 10.2.2), the complaint round can be made more efficient in both eJF-DKG and AMT DKG. Furthermore, we show AMT DKG players can verify their shares much faster under certain conditions (see Section 10.3.1). Finally, in Section 10.4.3, we show that our smaller dealing time more than makes up for these increases.

10.3.1 Fast-track Verification Round

During the verification round, each player j must receive and verify shares from all players $i \in [n]$, including himself (see Algorithm 8). Specifically, each player i gives j: (1) a KZG commitment c_i of i's polynomial f_i, (2) a share $s_{i,j} = f_i(\omega_N^{j-1})$ with an AMT proof $\pi_{i,j}$ and (3) $g^{f_i(0)}$ with a NIZKPoK and KZG proof. Next, player j must verify each $s_{i,j}$ and $g^{f_i(0)}$ against their c_i. With *naive verification*, this takes $\Theta(n \log t)$ pairings for all $s_{i,j}$'s (since $\pi_{i,j}$'s are AMT proofs), and $\Theta(n)$ pairings for the $g^{f_i(0)}$'s. We show how *batch verification* can do this faster, with anywhere from $\Theta(\log t)$ to $\Theta(n \log t)$ pairings, depending on the number of valid shares. (We will not address the $\Theta(n)$ work required to verify all NIZKPoKs.)

First, consider the *best case* when all $s_{i,j}$'s are valid. The key idea is player j will verify just one *aggregated* share $s_j = \sum_{i=1}^{n} s_{i,j}$ against an aggregated commitment $c_{\mathsf{all}} = \prod_{i=1}^{n} c_i$ and aggregated proof π_j from all $\pi_{i,j}$'s (as explained in Section 9.1.3). (We ignore the $g^{f_i(0)}$'s for now.) This takes $\Theta(n \log t)$ aggregation work but only takes $\Theta(\log t)$ pairings. If successful, j has a valid share s_j on $f_{\mathsf{all}} = \sum_{i=1}^{n} f_i$. The same aggregation can be done on the $g^{f_i(0)}$'s and their KZG proofs. This way, the number of pairings is reduced significantly to $\Theta(\log t)$ for the shares and $\Theta(1)$ for the $g^{f_i(0)}$'s. (Again, j still does $\Theta(n)$ work to verify the NIZKPoKs individually, which we will not address.)

Since players can be malicious, let us consider an *average case* when a small number of b shares are bad. In this case, j can identify the b shares faster via *batch verification* [28]. Specifically, j starts with the shares, proofs and commitments as leaves of a binary tree, where every node aggregates its subtree's shares, proofs and commitments. As a result, the root will contain $(c_{\mathsf{all}}, s_j, \pi_j)$. If verification of the root fails, j proceeds recursively down the tree. Whenever a node verifies, shares in its subtree will no longer be checked individually, saving work for j. In this fashion, j only computes $\Theta(b \log t)$ pairings if $\leq b$ shares are bad.

Unfortunately, in the *worst case* (i.e., $t-1$ bad shares), batch verification computes $\approx (2n-1) \log t$ pairings, which is slower than the $\approx n \log t$ pairings when done naively. Thus, as pointed out by previous work [138], j should abort and verify naively after too many nodes fail verification. To summarize, j can compute fewer pairings by batch-verifying optimistically to see if he is in the best or average case and downgrading to naive verification otherwise. We stress that j still does $\Theta(n \log t)$ work to build the tree and $\Theta(n)$ work to verify all NIZKPoKs, but fewer (expensive) pairings are computed.

10.3.2 Optimistic Reconstruction

DKG protocols have the advantage that g^s must be exposed to all players and the reconstructor. Thus, the reconstructor can optimistically interpolate s from any t shares (without verifying them) and check the result against g^s. In the best case, when all or most shares are valid, this will recover the correct s very fast (see Section 10.4.3.3). (Note that AMT VSS and eVSS do not expose g^s but they could be easily modified to do so and speed up the reconstruction in the best case, at a very small increase in dealing time.) In the worst case, AMT DKG's reconstruction time is the same as AMT VSS's (see Section 10.2.3).

10.4 Implementation

In this section, we demonstrate the scalability of our proposed cryptosystems. Our experiments focus on the difficult case when $t > n/2$, specifically $t = f + 1$ and $n = 2f + 1$. We benchmark TSS, VSS and DKG cryptosystems for thresholds $t \in \{2^1, 2^2, 2^3, \ldots, 2^{20}\}$. Although we did not benchmark other thresholds, similar performance gains would have been observed for other sufficiently large values of t (e.g., $t = f + 1$ and $n = 3f + 1$). However, we acknowledge that, for sufficiently small t, eVSS's and eJF-DKG's $\Theta(nt)$ dealing would outperform ours. Similarly, in this small t setting, naive Lagrange interpolation would outperform fast Lagrange. Our experiments show that:

- Our BLS TSS scales to $n \approx 2$ million signers and outperforms the naive scheme as early as $n = 511$ (see Figure 10-1).
- AMT VSS scales to hundreds of thousands of participants, and outperforms eVSS as early as $n = 255$ (see Figure 10-5).
- AMT DKG scales to $n \approx 65{,}000$ players and outperforms eJF-DKG at $n = 1023$ (see Figure 10-8).

Importantly, our VSS and DKG speed-ups come at the price of a modest increase in communication (see Figure 10-9). For example, for $n \approx 65{,}000$, a DKG player's communication during dealing increases by $4.11\times$ from 18 MiB in eJF-DKG to 74 MiB in AMT DKG. However, since the worst-case end-to-end time decreases by $32\times$ from 16.76 hrs in eJF-DKG to 30.83 mins in AMT DKG, the extra communication should be worth it in many applications.

For prohibitively-slow experiments with large t, we repeat them fewer times than experiments with smaller t. For brevity, we specify the amount of times we repeat an experiment for each threshold via a *measurement configuration*. For example, the measurement configuration of our efficient BLS threshold scheme is $\langle 7 \times 100, 13 \times 10 \rangle$. This means that for the first 7 thresholds $t \in \{2^1, 2^2, \ldots, 2^7\}$ we ran the experiment 100 times while for the last 13 thresholds we ran it 10 times.

Codebase and Experimental Setup. We implemented (1) our BLS threshold signature scheme from Section 10.1, (2) eJF-DKG [126] and AMT DKG and (3) eVSS [128] and AMT VSS in 5700 lines of C++. We used a 254-bit Barretto-Naehrig curve with a Type III pairing [15] from Zcash's `libff` [189] elliptic curve library. We used `libfqfft` [190] to multiply polynomials fast using FFT. All experiments were run on an Intel Core i7 CPU 980X @ 3.33GHz with 12 cores and 20 GB of RAM, running Ubuntu 16.04.6 LTS (64-bit version). Since all benchmarked schemes would benefit equally from multi-threading, we did not implement it.

Limitations. Our DKG and VSS evaluations do not account for network delays. This is an important limitation. Our focus was on the computational bottlenecks of these protocols. Nonetheless, scaling and evaluating the broadcast channel of VSS and DKG protocols is necessary, interesting future work. In particular, ideas from scalable consensus protocols [102] could be used for this. Finally, our VSS and DKG "worst case" evaluations do not fully account for malicious behavior. Specifically, they do not account for the additional communication

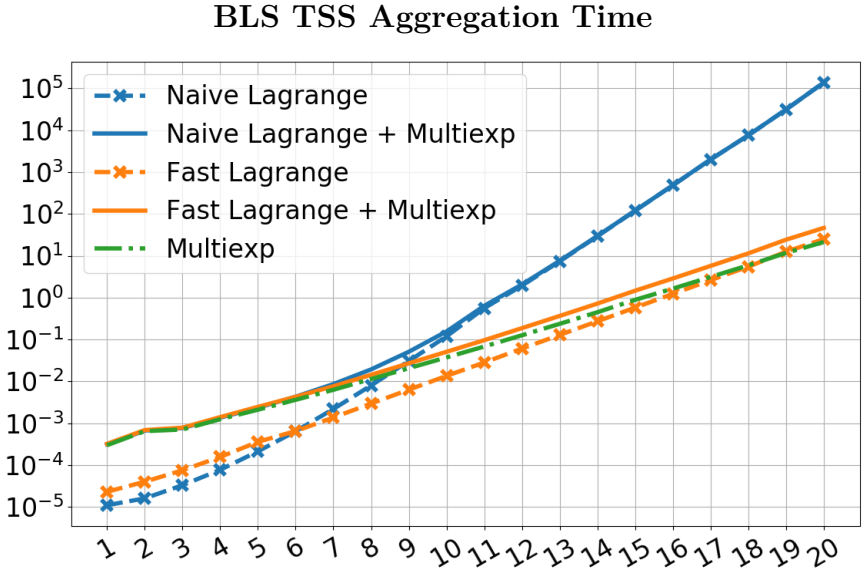


Figure 10-1: This figure plots the time to aggregate a (t, n) BLS TSS (for $t = f + 1$ and $n = 2f + 1$), excluding the share verification cost. The y axis is in seconds and the x axis is $\log_2 t$. We plot the individual cost of each aggregation phase: interpolation and multi-exponentiation. First, we see that, for BLS with *naive Lagrange*, the signature aggregation cost is dominated by the cost of naive Lagrange interpolation, as n gets larger. Second, we see that that our $\Theta(t \log^2 t)$ *fast Lagrange* interpolation beats the $\Theta(t^2)$ naive Lagrange interpolation as early as $t \geq 128$ (or $n \geq 255$). Third, as n gets larger, we see that BLS with fast Lagrange is orders of magnitude faster than with naive Lagrange.

and computational cost associated with complaint broadcasting. We hope to address this in future work (see Section 11.2.1.2).

10.4.1 BLS Threshold Signature Experiments

First, we sample a random subset of t signers T with valid signature shares $\{\sigma_i\}_{i \in T}$. Second, we compute Lagrange coefficients $\mathcal{L}_i^T(0)$ w.r.t. points $x_i = \omega_N^{i-1}$ (see Section 2.4) using both fast and naive Lagrange. Third, we compute the final threshold signature $\sigma = \prod_{i \in T} \sigma_i^{\mathcal{L}_i^T(0)}$ using a multi-exponentiation. The measurement configuration for fast Lagrange is $\langle 7 \times 100, 13 \times 10 \rangle$ while for naive Lagrange is $\langle 8 \times 100, 6 \times 10, 8, 4, 2, 1, 1, 1 \rangle$. We plot the average aggregation time in Figure 10-1 and observe that our scheme beats the naive scheme as early as $n = 511$. We do not measure the time to identify valid signature shares via batch verification [28], which our techniques leaves unchanged.

Our results show that our fast Lagrange interpolation drastically reduces the time to aggregate when $t \approx n/2$. Specifically, for $n \approx 2^{21}$, we aggregate a signature in 46.26 secs, instead of 1.59 days if aggregated via naive Lagrange (2964× faster). The benefits are not as drastic for smaller thresholds, but remain significant. For example, for $n \approx 2^{15}$, we reduce the time by 41× from 29.74 secs to 719.65 ms. For $n = 4095$, we see a 6.6× speed-up from 636.6 ms to 96.17 ms. For $n = 2047$, we see a 3× speed-up from 155.62 ms to 50.74 ms.

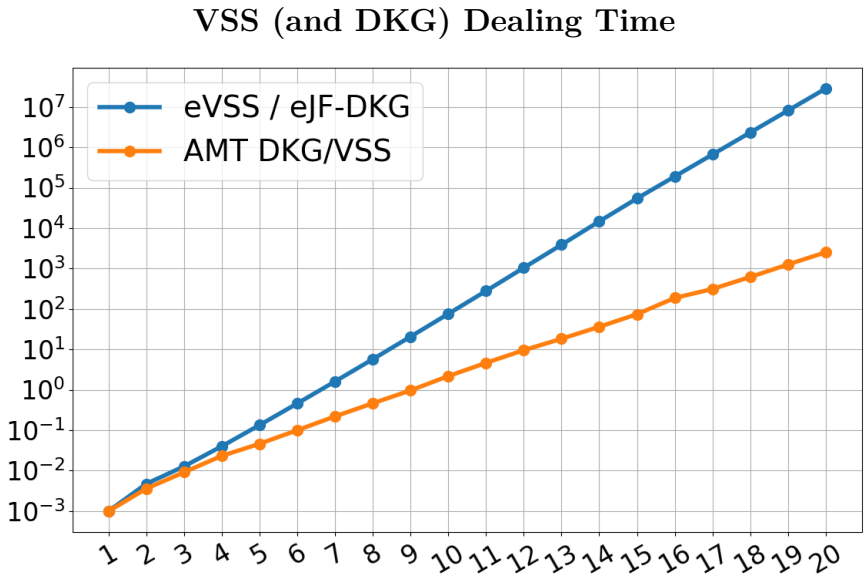


Figure 10-2: This figure shows that our AMT VSS/DKG dealing is orders of magnitude faster than eVSS and eJF-DKG dealing. The y axis is the dealing round time in seconds and the x axis is $\log_2 t$. We achieve this speed-up by replacing the $\Theta(tn)$ time KZG proofs with our $\Theta(n \log t)$ time AMT proofs.

10.4.2 Verifiable Secret Sharing Experiments

In this section, we benchmark eVSS and AMT VSS. We do not benchmark the complaint round since, when implemented with KZG batch proofs, it remains the same (see Section 10.2.1).

10.4.2.1 VSS Dealing

For eVSS dealing, the measurement configuration is $\langle 10 \times 10, 3, 2, 2, 1, 1, 0, 0, 0, 0, 0 \rangle$. For large $t \geq 2^{16}$, eVSS dealing is too slow, so we extrapolate it from the previous dealing time (i.e., we multiply by 3.5). For AMT VSS dealing, the measurement configuration is $\langle 12 \times 100, 50, 22, 10, 5, 3, 2, 1, 1 \rangle$. In eVSS, we compute the shares s_i "for free" as remainders of the $\phi(x)/(x - i)$ divisions. We plot the average dealing time in AMT VSS and eVSS as a function of n in Figure 10-2. Our results show that AMT VSS's $\Theta(n \log t)$ dealing scales much better than eVSS's $\Theta(nt)$ dealing. For example, for $n \approx 65,000$, eVSS takes 15.1 hrs while AMT VSS takes 1.24 mins. For very large $n \approx 2^{21}$, eVSS takes a prohibitive 330 days while AMT VSS takes 42 mins. We find that AMT VSS's dealing outperforms eVSS's as early as $n = 31$.

10.4.2.2 VSS Verification Round

In Figure 10-3, we plot the time for one player to verify its share. The measurement configuration is $\langle 20 \times 1000 \rangle$ for both schemes. In eVSS, verification requires two pairings and one exponentiation in $\mathbb{G}_1$, taking on average 2.15 ms. In AMT VSS, verification requires $\lfloor \log (t - 1) \rfloor + 1$ pairings and one exponentiation in $\mathbb{G}_1$, ranging from 2.07 ms ($n = 3$) to

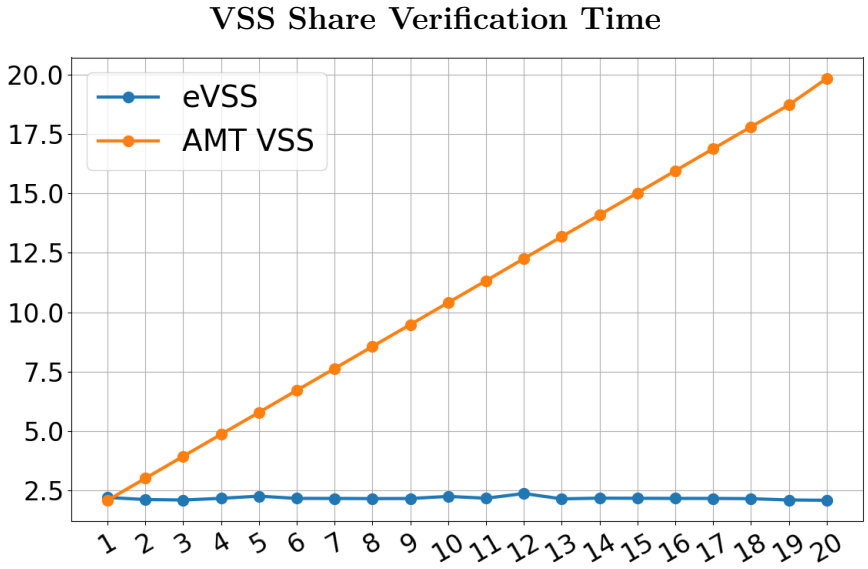


Figure 10-3: This graph plots the VSS verification round time for both eVSS and AMT VSS. The y axis is the verification round time in milliseconds and the x axis is $\log_2 t$. We can see AMT VSS's verification time is slower than eVSS (i.e., $\Theta(\log t)$ vs $\Theta(1)$ pairings). This is the price we pay for precomputing AMT proofs much faster.

19.85 ms ($n \approx 2^{21}$).

10.4.2.3 VSS Reconstruction

In Figure 10-4, we plot the time to reconstruct the secret. We consider the *best-case* and *worst-case* times, as detailed in Section 10.2.3. For eVSS, "best case" means the first t share verifications are successful and "worst case" means the first $n - t$ are unsuccessful (see Section 10.2.3). The measurement configuration is $\langle 5 \times 1000, 500, 250, 120, 60, 30, 15, 5 \times 10, 8, 4, 2, 1\rangle$ for eVSS and $\langle 9 \times 100, 4 \times 10, 4, 2, 5 \times 1\rangle$ for AMT VSS. In both protocols, the (fast) Lagrange interpolation time is insignificant compared to the time to verify shares during reconstruction (e.g., for $n \approx 2^{21}$ in eVSS, interpolation is only 25 secs out of the total 34 mins worst-case time).

AMT VSS's best-case is very close to eVSS's worst-case. This is because, with the help of memoization, AMT VSS's best case only computes $\leq 2n - 1$ pairings (i.e., the number of nodes in a full binary tree with n leaves). This closely matches the $2n$ pairings in eVSS's worst case. (In practice, we replace n of these pairings and $\mathbb{G}_1$ exponentiations by n $\mathbb{G}_T$ exponentiations, which are slightly faster.) AMT VSS's worst case is $1.12\times$ to $6\times$ slower than eVSS's. But we show next that our faster dealing more than makes up for this. Finally, eVSS's best-case time is half its worst-case time, as expected.

10.4.2.4 VSS End-to-End Time

Finally, we consider the *end-to-end time*, which is the sum of the sharing and reconstruction phase times. (Again, a limitation of our work is ignoring the overhead of the complaint round

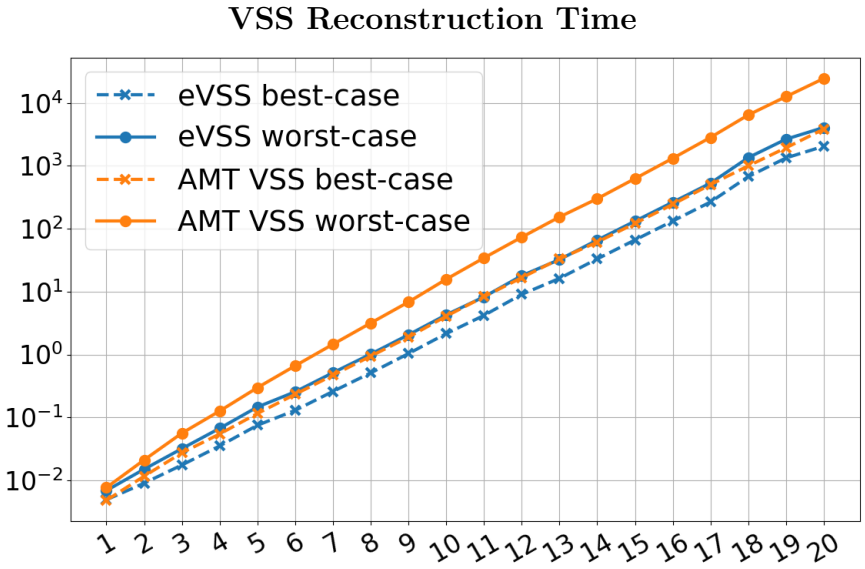


Figure 10-4: This graph shows that AMT VSS's reconstruction time is slower than in eVSS. The y axis is the time in seconds and the x axis is $\log_2 t$. An interesting observation is that AMT VSS's best-case reconstruction time is very close to eVSS's worst-case reconstruction time (we explain in Section 10.4.2.3 why).

in the worst case.) Figure 10-5 gives the best- and worst-case end-to-end times. The key takeaway is that AMT VSS's smaller dealing time makes up for the increase in its verification round and reconstruction phase times. AMT VSS outperforms eVSS's worst-case time at $n \geq 255$ and its best-case time at $n \geq 63$. For example, for large $n = 16,383$, we reduce the worst-case time from 1.1 hrs to 2.9 mins and the best-case time from 1.1 hrs to 51.48 secs. The best case improvement ranges from 1.26× ($n = 63$) to 4484× ($n \approx 2^{21}$). The worst case improvement ranges from 1.26× ($n = 255$) to 1055× ($n \approx 2^{21}$). Thus, we conclude AMT VSS scales better than eVSS.

10.4.3 Distributed Key Generation Experiments

Our DKG experiments mostly tell the same story as our VSS experiments: AMTs drastically reduce the dealing time of DKG players, which more than makes up for the slight increase in verification and reconstruction time. However, AMT DKG has a 1.2× to 5.2× communication overhead during dealing. Still, we believe this is worth the drastic reduction in end-to-end times (see Section 10.4.3.4).

10.4.3.1 DKG Dealing

DKG dealing time is equal to VSS dealing time (see Section 10.4.2.1) plus the time to compute a KZG proof and a NIZKPoK for $g^{f_i(0)}$. However, as n increases, the time to compute these two proofs pales in comparison to the time to compute the n evaluation proofs. Thus, in Figure 10-2, we treat DKG dealing times as equal to VSS dealing times. (Having separate VSS and DKG lines in Figure 10-2 would be pointless, as they would be almost indistinguishable.)

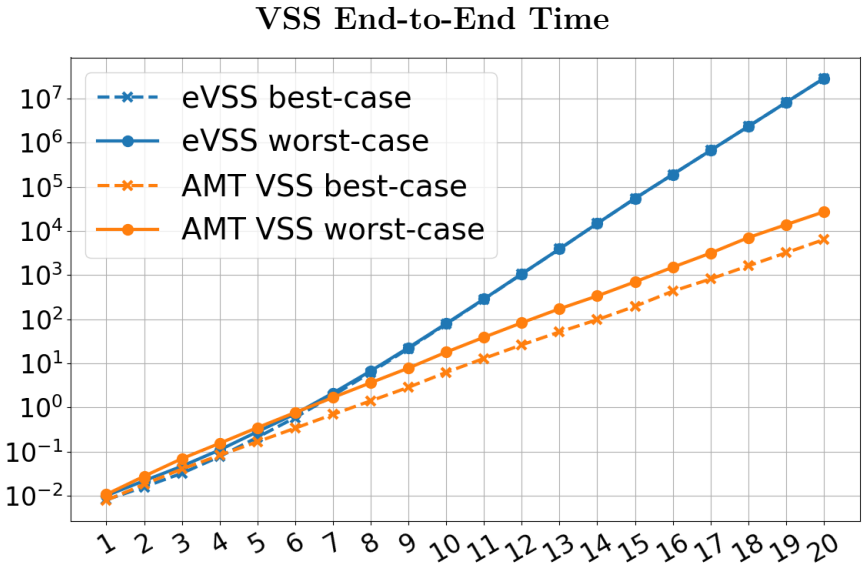


Figure 10-5: The y axis is the end-to-end time (i.e., sharing and reconstruction phases) in seconds and the x axis is $\log_2 t$. This graph shows that, in terms of the end-to-end execution time of the VSS protocol, AMT VSS is orders of magnitude faster than eVSS as n gets larger. This is despite our slower verification round and reconstruction phase, which are made up for by our much faster dealing round.

As a result, the same observations apply here as in Section 10.4.2.1: AMTs drastically reduce dealing times.

10.4.3.2 DKG Verification Round

We consider both the *best case* and the *worst case* verification time, as discussed in Section 10.3.1. In our best-case experiment, each player j aggregates all its shares as $s_j = \sum_{i \in [n]} s_{i,j}$ and their evaluation proofs as π_j. Then, j verifies s_j against π_j. Similarly, j aggregates and efficiently verifies all its $g^{f_i(0)}$'s and their KZG proofs. In the worst-case experiment, j individually verifies the $s_{i,j}$ shares and the $g^{f_i(0)}$'s. Importantly, in both experiments, j individually verifies all n NIZKPoKs for $g^{f_i(0)}$ in $\Theta(n)$ time. The two experiments are meant to bound the time of a realistic implementation that carefully uses *batch verification* [28, 138] to not exceed the worst-case time too much.

The best-case eJF-DKG measurement configuration is $\langle 8 \times 100, 50, 25, 12, 9 \times 10 \rangle$ and the worst-case is $\langle 5 \times 100, 50, 25, 12, 12 \times 10 \rangle$. For AMT DKG, the best-case configuration is $\langle 12 \times 100, 80, 40, 20, 16, 8, 4, 3, 2 \rangle$ and the worst-case is $\langle 5 \times 100, 4 \times 80, 40, 20, 8, 4, 2, 6 \times 1 \rangle$. The average per-player verification times are plotted in Figure 10-6. In the best case, both schemes perform roughly the same, since the verification of the n NIZKPoKs quickly starts dominating the aggregated proof verification. In the worst case, AMT DKG time ranges from 8.96 ms ($n = 3$) to 12.92 hrs ($n \approx 2^{21}$). In contrast, eJF-DKG time ranges from 8.92 ms to 2.59 hrs (1.5× to 5× faster). Nonetheless, eJF-DKG remains slower overall due to its much slower dealing (see Section 10.4.3.4). Both best- and worst-case times can be reduced by batch-verifying NIZKPoKs, which resemble Schnorr signatures [186] and are amenable to

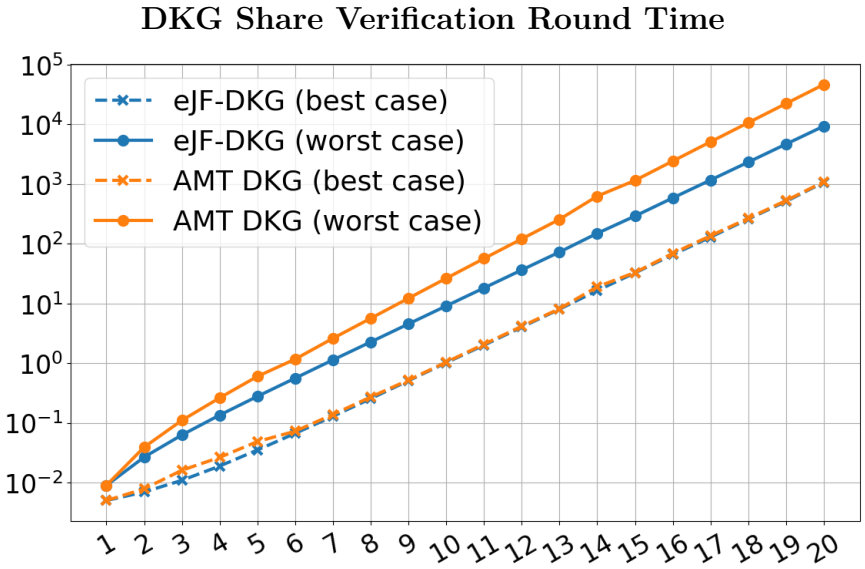


Figure 10-6: The y axis is the verification round time in seconds and the x axis is $\log_2 t$. This graph shows that AMT DKG's *worst-case* share verification round is slower than eJF-DKG's. This is due to our $\Theta(n \log t)$ cost to verify all n shares, compared to eJF-DKG's $\Theta(n)$ cost. However, in the *best case*, both protocols perform almost the same, since most of the time is spent aggregating proofs instead of verifying them.

batching [23].

10.4.3.3 DKG Reconstruction

Here the measurement configuration is $\langle 4 \times 1000, 200, 50, 25, 13 \times 10\rangle$ and times are plotted in Figure 10-7. The best case is very fast in both eJF-DKG and AMT DKG, taking only 24.71 secs for $t = 2^{20}$, since both schemes interpolate the secret s without verifying shares and check it against g^s (see Section 10.3.2). For the worst case, the time is the sum of (1) the (failed) best-case reconstruction time and (2) the worst-case time to identify t valid shares from n shares. Since the best case is very fast, the DKG worst-case time (see Figure 10-7) looks almost identical to its VSS counterpart (see Figure 10-4). Note that the same AMT VSS speed-up techniques for finding t valid shares apply in AMT DKG (see Section 10.2.3). AMT DKG's worst case is anywhere from 1.1× to 6× slower than eJF-DKG's, much like AMT VSS. However, as we show next, AMT DKG's faster dealing more than makes up for this.

10.4.3.4 DKG End-to-End Time

Similar to the VSS experiments in Section 10.4.2.4, we consider the *end-to-end time*. Figure 10-8 plots the best- and worst-case end-to-end times and shows that AMT DKG outperforms eJF-DKG starting at $n \geq 63$ (in the best case) and at $n \geq 1023$ (in the worst case). This is a direct consequence of AMT VSS outperforming eVSS, since the DKG protocols use these VSS protocols internally. For example, for large $n = 16,383$, we reduce the worst-case end-to-end time from 1.19 hrs to 7.12 mins and the best-case time from 1.16 hrs to 25.45 secs. The

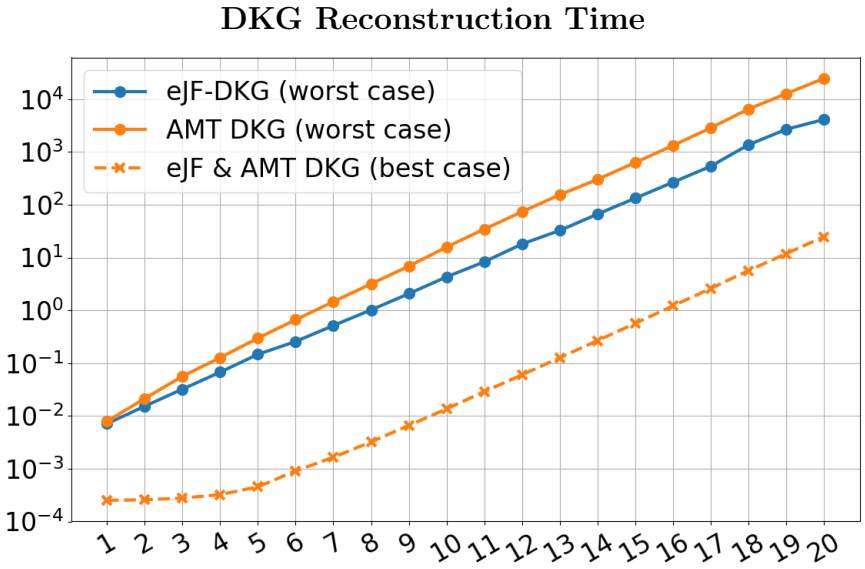


Figure 10-7: The y axis is the reconstruction phase time in seconds and the x axis is $\log_2 t$. This graph shows that, in the *best case*, both AMT DKG and eJF-DKG are just as fast, since they both just interpolate the secret s directly. In the *worst case*, however, AMT DKG has to spend more time verifying shares than eJF-DKG due to our larger proofs. Nonetheless, in Figure 10-8, we show our slower reconstruction time is worth it, since our end-to-end time is much smaller than in eJF-DKG.

improvement in best-case end-to-end time ranges from 1.6× ($n = 63$) to 8607× ($n \approx 2^{21}$) and, in the worst case, from 1.3× to 427×. Thus, we conclude AMT DKG scales better than eJF-DKG.

10.4.3.5 DKG Communication

We estimate each player's upload and download during the dealing round. For upload, each eJF-DKG and AMT DKG player i has to broadcast a KZG commitment $g^{f_i(\tau)}$ (32 bytes) and a commitment $g^{f_i(0)}$ with a NIZKPoK and a KZG proof (32 + 64 + 32 bytes). Then, i has to send each $j \in [n]$ its share (32 bytes) with an evaluation proof (32 bytes for KZG or $(\lfloor \log(t-1) \rfloor + 1) \cdot 32$ bytes for AMT). For download, each player i, has to download $n-1$ shares, each with their KZG commitment and evaluation proof, plus $n-1$ $g^{f_j(0)}$'s, each with their NIZKPoK and KZG proof. Note that AMT DKG uses KZG proofs for $g^{f_i(0)}$ to minimize its communication overhead.

We plot the upload and download numbers for both schemes in Figure 10-9. eJF-DKG's per-player upload ranges from 288 bytes to 128 MiB while download ranges from 448 bytes to 448 MiB. AMT VSS's upload overhead ranges from 1.0× to 10.5× and its download overhead ranges from 1.0× to 3.7×. Overall, AMT VSS's upload-and-download overhead ranges from 1.0× to 5.2×. Thus, we believe the 8607× and 427× reductions in best- and worst-case end-to-end times are sufficiently large to make up for this overhead.

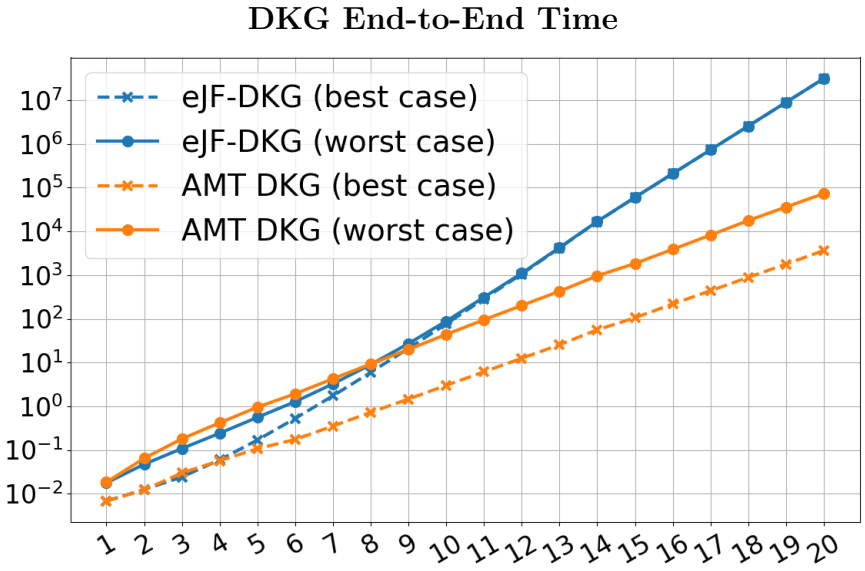


Figure 10-8: The y axis is the end-to-end time (i.e., sharing and reconstruction phases) in seconds and the x axis is $\log_2 t$. This graph shows that, in terms of the end-to-end execution time of the DKG protocol, AMT DKG is orders of magnitude faster than eJF-DKG as n gets larger. This is because our slower verification round and reconstruction phase are more than made up for by our much faster dealing round.

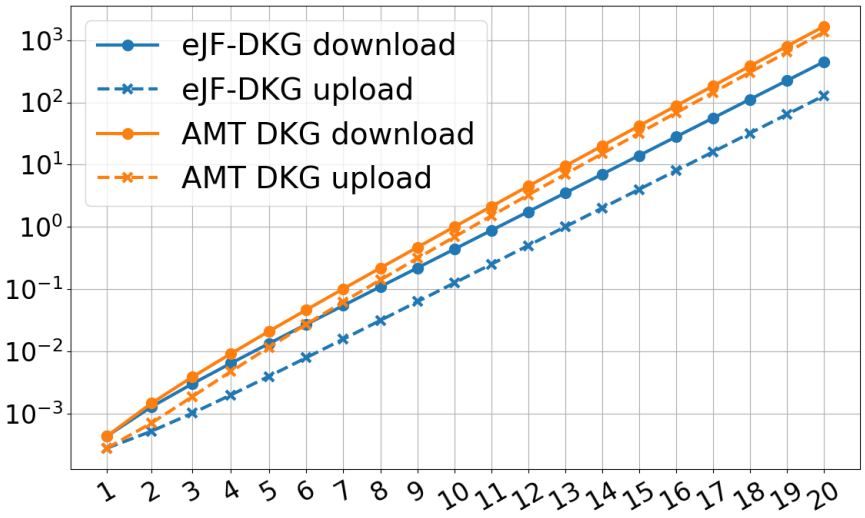


Figure 10-9: The y axis is the communication in mebibytes (MiBs) and the x axis is $\log_2 t$. This graph shows that AMT DKG incurs slightly higher communication than eJF-DKG. The per-player upload during dealing increases by at most 10.5× and the download by at most 3.7×. Overall, AMT DKG's communication overhead ranges from 1.0× to 5.2×.

Part III

Conclusion and Future Work

Chapter 11

Future Work

11.1 Append-only Authenticated Data Structures

11.1.1 Improving Our Current AADs

11.1.1.1 De-amortization

In Sections 5.3 and 6.3, we mentioned that our *amortized* append time can be de-amortized to a worst-case time using known, generic techniques for any static data structure [165, 166]. However, we did not investigate the details of how to do this *efficiently* in this thesis. We believe it would be interesting future work to optimize such a de-amortized construction.

11.1.1.2 Compressed Tries

A big source of overhead in our AAS and AAD are the "sparse" tries of height 2λ (see Sections 5.3 and 6.3.1) and 4λ (see Section 6.3.2), respectively. Replacing these with a compressed trie of height $h = O(\log n)$ would drastically reduce append times, server storage and server memory.

For now, let us focus on the AAS. Suppose we limit all trie heights to some arbitrary h (e.g., $h = 32$). At the same time, we redefine the frontier of such tries to consist, as before, of nodes that (1) are not in the trie but have parents in the trie and that (2) are at depth $\leq h$. (The root node is at depth 0 and a trie with $h = 0$ has just one node: the root.)

Immediately, we notice this design suffers from ambiguity. Consider two elements e_1, e_2 that have the same first h bits. Recall that these bits are used to determine a path in the trie. Unfortunately, since e_1 and e_2 have the same first h bits, this path cannot be used to tell which one of them was inserted. In fact, it could be that none of them was inserted (e.g., another element e_3 with the same first h bits was inserted.) In contrast, when the trie height was 2λ, no such ambiguity was possible, unless e_1 and e_2 had the same cryptographic hash, which would imply a collision in the CRHF. Nonetheless, we believe additional techniques can be used to disambiguate between such conflicting elements, at the cost of slightly increasing lookup proof size. We leave exploring this to future work.

11.1.1.3 Reducing Server Space

Our current approach for proving completeness for lookup proofs in $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{bilinear}}$ and $\mathsf{AAD}^{\mathsf{short}}_{\mathsf{rsa}}$ set requires FCTs at every node in the forest. This spikes storage to $O(\lambda n \log n)$ on the server, as opposed to $O(\lambda n)$ for our previous approach from Section 6.3.2. An interesting question is if we can avoid this space increase while maintaining $O(2\lambda n)$ public parameters by removing the need for FCTs at internal nodes in the forest.

A promising approach could be changing the prefix accumulators to be multisets rather than sets of prefixes. This would allow the server to prove that all m paths to a key k have been given by showing that the root prefix accumulator accumulates k's prefixes with multiplicity *exactly* equal to m. This approach will concretely increase the server's public parameters size to *exactly* $\ell = (2\lambda + 1)n$, within our $O(2\lambda n)$ upper bound. We leave efficiently precomputing such proofs (perhaps via AMTs; see Chapter 9) and determining the increase in public parameters on the client to future work. We believe this "prefix multiset accumulator" approach could also be used in combination with our "compressed trie" approach (see Section 11.1.1.2).

11.1.1.4 Speeding up Frontier Proofs via AMTs

Our frontier technique from Section 5.3.2.2 is just a precomputation technique. Specifically, it proves that for each prefix ρ at the frontier of a trie, ρ is *not* accumulated in the prefix accumulator corresponding to that trie. Let α denote the characteristic polynomial committed in that prefix accumulator. In other words, our frontier technique precomputes all proofs for $\alpha(\rho) \neq 0$ by computing proofs for $\phi(\rho) = 0$ and for $\gcd(\alpha, \phi) = 1$. However, in Chapter 9, we introduced an AMT technique that does exactly this, without resorting to disjointness proofs. Thus, we hope to use AMTs to improve our frontier proof sizes and computation times in future work.

11.1.1.5 Parallelizing RSA-based Accumulators

A big source of overhead in our RSA-based AAD is committing to RSA accumulators and witnesses. These commitments consist of an inherently unparallelizable exponentiation with a huge exponent. (Indeed, this difficulty to parallelize is leveraged to build *verifiable delay functions* [30, 177, 209].) Thus, an interesting direction would be to develop *parallelizable* RSA accumulators that support committing to sets and computing witnesses faster.

One avenue for doing this is an RSA-based polynomial commitment scheme. Recent work [45] shows how to build such constant-sized polynomial commitments with logarithmic-sized evaluation proofs from hidden-order groups (see Section 4.1). Importantly, unlike RSA accumulators, these RSA polynomial commitments seem to support parallelizing the computation of a commitment. This could significantly speed up our RSA-based AAD.

11.1.1.6 AADs from The Generic Group Model

Our $\mathsf{AAD}^{\mathsf{tall}}_{\mathsf{bilinear}}$ implementation using Type III pairings incurs significant overhead from having to compute extractable counterparts of elements. It is worth exploring if hierarchical accumulator-based constructions such as our Communion Tree (CT) techniques have security

proofs that do not require these extractable counterparts. One such construction is our AMTs from Chapter 9 which, although hierarchical, can be proven secure under q-SBDH. We leave it to future work to find such security proofs, perhaps in the generic group model.

11.1.2 New AAD Constructions

11.1.2.1 Lower bounds for CRHF-based AAS

Before exploring new constructions, we should better understand what can(not) be done with simple cryptographic tools such as *collision-resistant hash functions* (CRHFs). We hope to prove lower bounds about what can be achieved with CRHFs in future work.

11.1.2.2 AADs from Lattices

It would be theoretically-interesting to construct AADs from lattice-based cryptography [175]. In particular, we believe modifications of Ajtai-based hash functions [6, 107] might prove useful for this. For example, the hash function proposed by Papamanthou et al. [168] has enough algebraic structure to give rise to a cryptographic accumulator. Thus, if enhanced with a subset witness, it could be used to build an AAD. We leave this to future work.

11.1.2.3 AADs from Argument Systems

A promising direction for future work is to build AADs from generic argument systems [11, 19, 20, 44, 95, 111, 112, 151, 208]. Such AAD constructions would also require non-standard assumptions [101], possibly different than q-PKE (e.g., random oracle model, generic group model, etc.). Depending on the argument system, they might or might not require trusted setup and large public parameters.

A static AAD can be built from an argument system (e.g., a SNARK [95, 112]) as follows. The AAD maintains one unsorted tree U and one sorted tree S whose leaves are sorted by key. The digest of the AAD consists of (1) the Merkle roots $(d(S), d(U))$ of S and U and (2) a SNARK *proof of "correspondence"* $\pi(S, U)$ between S and U. This proof shows that S's leaves are the same as U's *but in a different, sorted by key, order*. The SNARK circuit takes as input U's leaves and S's leaves, hashes them to obtain $d(U)$ and $d(S)$ and checks that S's leaves are sorted by key.

Now, given a digest $(d(S), d(U), \pi(S, U))$, lookups can be efficiently proven using Merkle proofs in the sorted tree S. The append-only property of two digests $(d(S), d(U), \pi(S, U))$ and $(d(S'), d(U'), \pi(S', U'))$ can be efficiently proven using an append-only proof between $d(U)$ and $d(U')$, since U and U' are just history trees [64]. This proves U is a subset of U' and, crucially, it also proves that S is a subset of S', since the SNARK $\pi(S, U)$ proves that S "corresponds" to U and S' to U'. Unfortunately, updates would invalidate the SNARK proof and would require time at least linear in the dictionary size to recompute this proof. However, we can apply the same Overmars technique [165, 166] to make updates polylogarithmic time. (This would now require a family of circuits, one for each size 2^i of the trees.)

This approach would result in much shorter lookup proofs while maintaining the same efficiency for append-only proofs, since state-of-the-art SNARKs have proofs consisting of just 3 group elements [112]. On the other hand, this approach might need more public parameters

and could have slower appends. This is because, even with SNARK-friendly hashes (e.g., Ajtai-based [21], MiMC [8] or Jubjub [214]), we estimate the number of multiplication gates for hashing trees of size 2^{20} to be in the billions. (And we are not accounting for the gates that verify tree S is sorted.) In contrast, the degrees of the polynomials in our bilinear-based constructions are only in the hundreds of millions for dictionaries of size 2^{20}.

Nonetheless, optimizing such a solution would be interesting future work. For example, replacing SNARKs with STARKs [19] would eliminate the large public parameters and the trusted setup, at the cost of larger append-only proofs. This may well be worth it if the proof size and prover time are not too large. Other argument systems such as Hyrax [208], Ligero [11] and Aurora [20] could achieve the same result. Unfortunately, Aurora and Ligero would increase the append-only proof verification time to linear, which could be prohibitive. Bulletproofs [44] would further increase this verification time to quasilinear. Hyrax can make this time sublinear if the circuit is sufficiently parallel or has "a wiring pattern [that] satisfies a technical regularity condition" [208].

Recursively-Composable Arguments. Another interesting approach is to obtain AADs from recursively-composable SNARKs [21, 25, 39, 60]. Such SNARKs could structure the verification of the append-only property recursively so that circuits need not operate on the entire dictionary, thus lowering overheads. We are aware of concurrent work that explores this approach, but unfortunately it is not peer-reviewed nor published in an online archive. While such an approach could be very promising, currently implemented systems operate at the 80-bit security level. This is because increasing the security of the elliptic curves used in recursive SNARK constructions is costly, since they have low embedding degree [21]. In contrast, our implementation is 100-bit-secure after accounting for recent progress on computing discrete logs [148] and our q-SDH assumption with $q = 2^{20}$ [29]. We can increase this to 118 bits, with no loss in performance, by adopting 128-bit-secure BLS12-381 curves [36].

11.2 Threshold Cryptosystems

11.2.1 Further Scaling VSS and DKG

11.2.1.1 Scaling the Broadcast Channel

A key bottleneck in VSS and DKG protocols is the broadcast channel used to agree on commitments to the polynomials that encode the secret(s). Future work can approach scaling the broadcast channel in two ways. The first way is to assume the broadcast channel is external to the VSS or DKG protocol and implemented by a different, smaller set of $N \ll n$ parties via a consensus protocol [53]. For example, ETHDKG [185] takes this approach by using the Ethereum blockchain [211] as the broadcast channel of a Feldman-style DKG protocol [99]. The second way is to assume an honest majority among the n VSS/DKG players and rely on this majority to securely run a synchronous consensus protocol [3]. Sortition techniques from scalable consensus protocols [102] can be employed to further scale such protocols.

11.2.1.2 Scaling the DKG Complaint Round

The DKG complaint round has worst-case computational quadratic overhead. Furthermore, in the worst case, it requires $O(f)$ broadcasts, each $O(f)$-sized, where f is the number of malicious players. This seriously limits the scalability of our DKG protocols in the worst-case setting of $f = t - 1$ malicious players.

Kate et al. [128] partially address this problem for VSS, not DKG, protocols via *batch proofs* as discussed in Section 10.2.2. Kate later applies batch proofs to the complaint round in DKG protocols too [126]. This is a step in the right direction. Even though batch proofs increase asymptotic computational complexity, they decrease the concrete computational complexity and the concrete communication complexity (see Sections 8.4.1 and 8.5.1).

Any future work that improves computing and verifying batch proofs will also improve the concrete complexity of DKG complaints. In general, any future work that reduces the complexity of the complaint round would be very interesting. For example, we explain next how running the DKG with randomly-picked *subcommittees* can do exactly this.

11.2.1.3 Sortitioned DKG

To further reduce communication and computation, we propose a *sortitioned DKG* where only a small, random *committee* of $c < n$ players deal. The key question is where does the randomness to pick the committee come from?

When a DKG runs many times, this randomness could come from previous DKG runs (e.g., DKGs for Schnorr TSS nonces). To bootstrap securely, the first DKG run would be with a full committee of size $c = n$.

When a DKG runs only once, such as when distributing the secret key of a (t, n) TSS, the c players could be a decentralized cothority [197] different than the TSS signers. The cothority would run the DKG dealing round while the n signers would run the DKG verification round (see Algorithm 8). The complaint round would be split: accused cothority members would compute the KZG batch proofs (see Section 10.2.2) while the n signers would receive and verify those proofs. Importantly, our AMT technique would help cothority members deal much more efficiently to the n signers by reducing both the required time and communication. We leave defining and proving the security of sortitioned DKGs to future work.

11.2.1.4 Scaling VSS and DKG in the Asynchronous Setting

A fascinating problem is to scale VSS and DKG in the *asynchronous setting* which assumes very little about message delivery. Such protocols sometimes make use of bivariate polynomials [125–127], for which constant-sized commitments with small-sized evaluation proofs are not known to exist. It would be interesting to come up with such a commitment scheme, perhaps modifying the new polynomial commitments based on RSA [45]. It would also be interesting to see if our AMT technique can be applied to bivariate polynomials to save computation in asynchronous VSS/DKG.

11.2.2 Enhancing AMTs

11.2.2.1 AMTs for Arbitrary Evaluation Points

AMTs can be generalized to any set of points $\{x_i\}_{i\in[n]}$ (not just $x_i = \omega_N^{i-1}$) *for which verifiers do not have the necessary accumulator commitments*. The accumulators $g^{a_w(\tau)}$ can be included as part of the proof *but* along with (1) a subset witness w.r.t. the parent accumulator and (2) an "extractable" counterpart $g^{\alpha a_w(\tau)}$, where α is another trapdoor. The asymptotic proof size remains the same but will increase in practice by 4x (with Type III pairings). Furthermore, this construction will need extra public parameters of the form $(g^{\alpha\tau^i})_{i\in[0,\ell]}$. On the other hand, proof verifiers now need $\Theta(1)$ rather than $\Theta(\log n)$ public parameters (see Sections 9.1.6 and 10.2.4). We leave proving this construction secure under ℓ-PKE [111] to future work.

11.2.2.2 Information-Theoretic Hiding AMTs

We can devise an information-theoretic hiding version of our AMT proofs that is compatible with information-theoretic hiding KZG commitments [128]. This version of AMTs can be used to speed up the unbiasable $\mathsf{New\text{-}DKG}$ protocol [99]. Let $h = g^{\kappa}$ be another generator of $\mathbb{G}$ such that nobody knows the discrete log $\kappa = \log_g(h)$. Assume that, in addition to $\mathsf{PP}_q(g;\tau)$, we also have public parameters $\mathsf{PP}_\ell(h;\tau)$.

An information-theoretic hiding KZG commitment to ϕ of degree d is $c = g^{\phi(\tau)}h^{r(\tau)} = g^{\phi(\tau)+\kappa r(\tau)}$ where r is a random, degree d polynomial [128]. Note that c is just a commitment to the polynomial $\psi(x) = \phi(x) + \kappa r(x)$. As a consequence, all we have to do is build an AMT for ψ. For this, we compute an AMT for ϕ with public parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(g;\tau)$ and one for r but with parameters $\mathsf{PP}_\ell^{\mathsf{SDH}}(h;\tau)$. By homomorphically combining these two AMTs we get exactly the AMT for ψ (see Section 9.1.3). We leave proving this construction is information-theoretic hiding to future work.

11.2.2.3 Verifying an AMT with $\approx 2n-1$ Pairings

The efficient reconstruction techniques from Section 10.2.3 reduced the number of pairings when verifying an AMT, but still required $\Theta(t + (n-t)\log t)$ pairings in the worst case. At the cost of doubling the prover time and proof size, this can be reduced to $\approx 2n-1$ pairings, independent of how many proofs are valid. The key idea is to also include commitments to the *remainder polynomials* from the multipoint evaluation tree in the AMT (see Figure 2-1). This way, an entire AMT tree can be verified node-by-node, top-to-bottom by checking that the division at each node is correct. We leave proving this approach secure to future work.

Chapter 12

Conclusion

This thesis makes progress on two fundamental problems in cryptography: keeping secrets and exchanging public keys.

First, we showed how to scale verifiable secret sharing (VSS) to millions of participants. Our scalable VSS protocol can be used to make leaking a secret key much harder by splitting it across millions of devices and forcing adversaries to compromise more than half of them. We then showed how to use such a secret to *quickly* sign messages only if more than half of the million signers cooperate. Specifically, we proposed a threshold signature scheme (TSS) that scales to millions of signers. Compared to previous TSSs at this scale, our scheme can sign a message in tens of seconds rather than tens of hours. Our TSS can be used to decentralize signing authorities on the Internet and to create randomness beacons. Yet to securely bootstrap our scalable TSS, we need a scalable distributed key generation (DKG) to securely generate its key without anybody learning it. For this, we showed how to use our scalable VSS to obtain a scalable DKG. At the same time, we left some problems to future work, such as the VSS/DKG broadcast channel and the worst-case overhead of complaints in DKGs.

Second, we showed how to build append-only logs using only a single untrusted server and no additional trust assumptions. Our construction is the first to achieve logarithmic proof sizes for both append-only and lookup proofs. Our logs can be used to secure the public key infrastructures that are crucial for secure web browsing and for end-to-end encrypted messaging. Furthermore, our logs can help secure the currently-feeble software distribution ecosystem, where users can be easily tricked into downloading malicious binaries. Unfortunately, our log's polylogarithmic (de)amortized append time is not yet fast enough for practice and we hope to speed it up in future work. Similarly, (some of) our concrete proof sizes could also be further reduced in future work.

At the core of almost all constructions in this thesis lie new techniques for computing proofs fast in constant-sized polynomial commitments and in cryptographic accumulators. We believe some of these techniques could be of independent interest. For example, our authenticated multipoint evaluation tree (AMT) gives rise to a new vector commitment (VC) scheme, which can be used to scale cryptocurrencies.

Part IV

Appendix

Appendix A

Appendix

A.1 Polylogarithmic DKG Configurations

As discussed in Section 7.1, Canny and Sorkin presented a *sparse matrix* DKG with $O(m^3)$ time and communication per player, where m is a group size [51]. Depending on the desired threshold (t, n) and the number f of malicious nodes tolerated, the group size m can be as small as $\Theta(\log^3(n))$. Unfortunately, for f sufficiently close to t, the group size becomes too large, approaching $n/2$ (see Table A.1).

Table A.1: The group size m and for various (t, n) sparse matrix DKGs with f failures tolerated.

ε	Group size m	$f = (1/2 - \varepsilon)n$	$t = (1/2 + \varepsilon)n$	n
0.1	50,598	26,214	39,321	65,536
0.15	14,222	22,937	42,598	65,536
0.2	5,691	19,660	45,875	65,536
0.25	2,735	16,384	49,152	65,536
0.3	1,475	13,107	52,428	65,536
0.33	1,057	11,141	54,394	65,536
0.4	505	6,553	8,982	65,536
0.05	445,909	471,859	576,716	1,048,576
0.1	53,022	419,430	629,145	1,048,576
0.15	14,949	367,001	681,574	1,048,576
0.2	5,977	314,572	734,003	1,048,576
0.25	2,871	262,144	786,432	1,048,576
0.3	1,548	209,715	838,860	1,048,576
0.33	1,107	178,257	870,318	1,048,576
0.4	527	104,857	943,718	1,048,576

A.2 Threshold Cryptosystems Performance Numbers

Total # of players	Dealing round time	Verification round time	Reconstruction phase time (BC / WC)		End-to-end time (BC / WC)		End-to-end speed-up (BC / WC)	
$n = 3$	989 mus 987 mus	2.19 ms 2.07 ms	4.9 ms 4.8 ms	6.9 ms 7.6 ms	8.1 ms 7.86 ms	10 ms 10.7 ms	1.03×	0.94×
$n = 7$	4.7 ms 3.6 ms	2.11 ms 3.00 ms	9 ms 11.6 ms	15 ms 21 ms	15.8 ms 18.19 ms	21.8 ms 27 ms	0.87×	0.79×
$n = 15$	12.7 ms 9.1 ms	2.09 ms 3.93 ms	17.5 ms 27 ms	31.9 ms 56 ms	32 ms 40.33 ms	46 ms 69 ms	0.8×	0.67×
$n = 31$	39.9 ms 23 ms	2.16 ms 4.86 ms	35 ms 54 ms	67 ms 126 ms	77 ms 82.46 ms	109 ms 154 ms	0.94×	0.71×
$n = 63$	133 ms 45.9 ms	2.25 ms 5.78 ms	75 ms 115 ms	146 ms 297 ms	210 ms 167.12 ms	282 ms 349 ms	1.26×	0.81×
$n = 127$	457 ms 99.5 ms	2.15 ms 6.71 ms	129 ms 233 ms	255 ms 662 ms	589 ms 339.30 ms	715 ms 768.5 ms	1.7×	0.93×
$n = 255$	1.6 secs 219 ms	2.15 ms 7.63 ms	259 ms 469 ms	513 ms 1.45 secs	1.9 secs 696.11 ms	2.1 secs 1.68 secs	2.7×	1.26×
$n = 511$	5.7 secs 461 ms	2.15 ms 8.56 ms	510 ms 941 ms	1 sec 3.2 secs	6.2 secs 1.41 secs	6.7 secs 3.64 secs	4.4×	1.85×
$n = 1023$	20.6 secs 963 ms	2.15 ms 9.48 ms	1 sec 1.9 secs	2.06 secs 6.8 secs	21 secs 2.86 secs	22.7 secs 7.8 secs	7.6×	2.91×
$n = 2047$	1.26 mins 2.2 secs	2.24 ms 10.40 ms	2.1 secs 4 secs	4.28 secs 15.7 secs	1.3 mins 6.25 secs	1.33 mins 18 secs	12×	4.45×
$n = 4095$	4.65 mins 4.6 secs	2.16 ms 11.33 ms	4.2 secs 8 secs	8.25 secs 34 secs	4.7 mins 12.94 secs	4.8 mins 39 secs	21×	7.36×
$n = 8191$	17.4 mins 9.5 secs	2.36 ms 12.26 ms	9 secs 16 secs	17.9 secs 1.2 mins	17.6 mins 26.24 secs	17.7 mins 1.4 mins	40×	13×
$n = 16383$	1.1 hrs 18 secs	2.13 ms 13.19 ms	16 secs 33 secs	32 secs 2.5 mins	1.1 hrs 51.48 secs	1.1 hrs 2.9 mins	75×	23×
$n = 32767$	4.1 hrs 36 secs	2.17 ms 14.10 ms	33 secs 1 mins	1.1 mins 5 mins	4.1 hrs 1.62 mins	4.1 hrs 5.6 mins	151×	43×
$n = 65535$	15.1 hrs 1.24 mins	2.16 ms 15.03 ms	1.1 mins 2 mins	2.20 mins 10.5 mins	15 hrs 3.27 mins	15 hrs 11.7 mins	277×	77×
$n = 131071$	2.2 days 3.1 mins	2.15 ms 15.96 ms	2.2 mins 4.1 mins	4.41 mins 22 mins	2.2 days 7.20 mins	2.2 days 25 mins	441×	126×
$n = 262143$	7.7 days 5.2 mins	2.15 ms 16.89 ms	4.5 mins 8.4 mins	8.84 mins 47 mins	7.7 days 13.60 mins	7.7 days 52 mins	817×	212×
$n = 524287$	27 days 10.3 mins	2.14 ms 17.81 ms	11 mins 16.6 mins	22.5 mins 1.8 hrs	27 days 26.96 mins	27 days 1.96 hrs	1442×	330×
$n = 1048575$	94 days 21 mins	2.09 ms 18.73 ms	22 mins 32 mins	44 mins 3.5 hrs	94 days 53.41 mins	94 days 3.86 hrs	2548×	588×
$n = 2097151$	330 days 42 mins	2.08 ms 19.85 ms	34 mins 1.07 hrs	1.13 hrs 6.8 hrs	330 days 1.77 hrs	330 days 7.5 hrs	4484×	1055×

Table A.2: Detailed performance numbers from Section 10.4.2 for eVSS vs. AMT VSS.

Total # of players	Dealing round time	Verification round time (best case / worst case)	Reconstruction phase time (best case / worst-case)	End-to-end time (best case / worst-case)
$n = 3$	1.57 ms	4.99 ms / 8.92 ms	252.00 mus / 7.12 ms	6.81 ms / 17.60 ms
$n = 3$	1.50 ms	4.98 ms / 8.96 ms	252.00 mus / 7.91 ms	6.74 ms / 18.36 ms
$n = 7$	5.26 ms	6.92 ms / 26.63 ms	258.00 mus / 15.28 ms	12.44 ms / 47.17 ms
$n = 7$	4.11 ms	7.83 ms / 39.05 ms	258.00 mus / 21.34 ms	12.20 ms / 64.50 ms
$n = 15$	13.03 ms	10.88 ms / 62.59 ms	278.00 mus / 32.17 ms	24.18 ms / 107.79 ms
$n = 15$	13.54 ms	16.03 ms / 109.90 ms	278.00 mus / 56.46 ms	29.85 ms / 179.90 ms
$n = 31$	40.06 ms	18.84 ms / 134.61 ms	323.00 mus / 67.40 ms	59.22 ms / 242.07 ms
$n = 31$	30.11 ms	26.34 ms / 261.69 ms	323.00 mus / 126.69 ms	56.77 ms / 418.50 ms
$n = 63$	133.71 ms	35.01 ms / 277.07 ms	454.00 mus / 146.86 ms	169.17 ms / 557.64 ms
$n = 63$	57.98 ms	48.48 ms / 598.95 ms	454.00 mus / 297.71 ms	106.92 ms / 954.64 ms
$n = 127$	459.12 ms	66.51 ms / 559.60 ms	910.00 mus / 256.33 ms	526.54 ms / 1.28 secs
$n = 127$	100.81 ms	72.25 ms / 1.17 secs	910.00 mus / 663.19 ms	173.97 ms / 1.93 secs
$n = 255$	1.60 secs	128.56 ms / 1.13 secs	1.65 ms / 514.93 ms	1.73 secs / 3.25 secs
$n = 255$	212.78 ms	137.87 ms / 2.60 secs	1.65 ms / 1.46 secs	352.29 ms / 4.27 secs
$n = 511$	5.70 secs	254.13 ms / 2.26 secs	3.23 ms / 1.02 secs	5.96 secs / 8.97 secs
$n = 511$	450.77 ms	270.25 ms / 5.62 secs	3.23 ms / 3.17 secs	724.24 ms / 9.24 secs
$n = 1023$	20.61 secs	507.00 ms / 4.53 secs	6.58 ms / 2.07 secs	21.12 secs / 27.20 secs
$n = 1023$	937.16 ms	519.82 ms / 12.19 secs	6.58 ms / 6.84 secs	1.46 secs / 19.96 secs
$n = 2047$	1.25 mins	1.01 secs / 9.08 secs	13.59 ms / 4.29 secs	1.27 mins / 1.48 mins
$n = 2047$	1.95 secs	1.04 secs / 26.33 secs	13.59 ms / 15.77 secs	3.00 secs / 44.04 secs
$n = 4095$	4.75 mins	2.00 secs / 18.14 secs	28.81 ms / 8.28 secs	4.79 mins / 5.19 mins
$n = 4095$	4.04 secs	2.07 secs / 56.44 secs	28.81 ms / 34.42 secs	6.14 secs / 1.58 mins
$n = 8191$	17.53 mins	4.00 secs / 36.34 secs	60.01 ms / 17.99 secs	17.60 mins / 18.44 mins
$n = 8191$	8.36 secs	4.14 secs / 2.01 mins	60.01 ms / 1.23 mins	12.55 secs / 3.37 mins
$n = 16383$	1.16 hrs	8.01 secs / 1.21 mins	126.44 ms / 32.50 secs	1.16 hrs / 1.19 hrs
$n = 16383$	17.18 secs	8.15 secs / 4.27 mins	126.44 ms / 2.57 mins	25.45 secs / 7.12 mins
$n = 32767$	4.63 hrs	16.41 secs / 2.49 mins	266.55 ms / 1.10 mins	4.64 hrs / 4.69 hrs
$n = 32767$	36.92 secs	19.00 secs / 10.44 mins	266.55 ms / 5.01 mins	56.18 secs / 16.07 mins
$n = 65535$	16.64 hrs	32.46 secs / 4.85 mins	565.44 ms / 2.21 mins	16.65 hrs / 16.76 hrs
$n = 65535$	1.20 mins	32.96 secs / 19.15 mins	565.44 ms / 10.47 mins	1.76 mins / 30.83 mins
$n = 131071$	2.43 days	1.09 mins / 9.74 mins	1.21 secs / 4.43 mins	2.43 days / 2.44 days
$n = 131071$	2.48 mins	1.15 mins / 40.39 mins	1.21 secs / 22.01 mins	3.65 mins / 1.08 hrs
$n = 262143$	8.49 days	2.15 mins / 19.43 mins	2.59 secs / 8.88 mins	8.49 days / 8.51 days
$n = 262143$	5.03 mins	2.26 mins / 1.42 hrs	2.59 secs / 47.28 mins	7.34 mins / 2.29 hrs
$n = 524287$	29.72 days	4.32 mins / 38.82 mins	5.64 secs / 22.64 mins	29.73 days / 29.77 days
$n = 524287$	10.26 mins	4.44 mins / 2.96 hrs	5.64 secs / 1.79 hrs	14.79 mins / 4.93 hrs
$n = 1048575$	104.03 days	8.52 mins / 1.30 hrs	11.78 secs / 44.51 mins	104.04 days / 104.11 days
$n = 1048575$	20.85 mins	8.80 mins / 6.19 hrs	11.78 secs / 3.51 hrs	29.85 mins / 10.05 hrs
$n = 2097151$	364.10 days	17.84 mins / 2.59 hrs	24.71 secs / 1.14 hrs	364.12 days / 364.26 days
$n = 2097151$	42.35 mins	18.15 mins / 12.92 hrs	24.71 secs / 6.83 hrs	1.02 hrs / 20.45 hrs

Table A.3: Detailed performance numbers from Section 10.4.3 for eJF-DKG vs. AMT DKG.

Bibliography

[1] Masayuki Abe and Serge Fehr. Adaptively Secure Feldman VSS and Applications to Universally-Composable Threshold Cryptography. In Matt Franklin, editor, *CRYPTO 2004*, pages 317–334, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[2] Harold Abelson, Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G Neumann, et al. Keys under doormats: mandating insecurity by requiring government access to all data and communications. *Journal of Cybersecurity*, 1(1):69–79, 2015.

[3] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine Agreement with Expected O(1) Rounds, Expected $O(n^2)$ Communication, and Optimal Resilience. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, pages 320–334, Cham, 2019. Springer International Publishing.

[4] Heather Adkins. An update on attempted man-in-the-middle attacks. `http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html`, 2011. Accessed: 2015-08-22.

[5] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[6] M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery.

[7] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A Practical System for Binary Transparency. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2018.

[8] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *ASIACRYPT'16*, 2016.

[9] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *USENIX ATC'16*, 2016.

[10] Salim Ali Altug and Yilei Chen. A Candidate Group with Infeasible Inversion. Cryptology ePrint Archive, Report 2018/926, 2018. `https://eprint.iacr.org/2018/926`.

[11] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM CCS'17*, 2017.

[12] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *Information Security*, 2001.

[13] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 913–930, New York, NY, USA, 2018. ACM.

[14] Niko Barić and Birgit Pfitzmann. Collision-Free Accumulators and Fail-Stop Signature Schemes Without Trees. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 480–494, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[15] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *Selected Areas in Cryptography*, 2006.

[16] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: Attack Resilient Public-Key Infrastructure. In *ACM CCS'14*, 2014.

[17] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2387–2402, New York, NY, USA, 2019. ACM.

[18] Mihir Bellare and Adriana Palacio. The Knowledge-of-Exponent Assumptions and 3-Round Zero-Knowledge Protocols. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, pages 273–289, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/2018/046`.

[20] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent Succinct Arguments for R1CS. In *EUROCRYPT'19*, 2019.

[21] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable Zero Knowledge Via Cycles of Elliptic Curves. *Algorithmica*, 79(4), Dec 2017.

[22] Josh Benaloh and Michael de Mare. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *EUROCRYPT'93*, 1994.

[23] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012.

[24] J. Berrut and L. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

[25] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive Composition and Bootstrapping for SNARKS and Proof-carrying Data. In *STOC'13*, 2013.

[26] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. On the Existence of Extractable One-way Functions. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 505–514, New York, NY, USA, 2014. ACM.

[27] G. R. Blakley. Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*, pages 313–318, June 1979.

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

[29] Dan Boneh and Xavier Boyen. Short Signatures Without Random Oracles and the SDH Assumption in Bilinear Groups. *Journal of Cryptology*, 21(2):149–177, Apr 2008.

[30] Dan Boneh, Benedikt Bünz, and Ben Fisch. A Survey of Two Verifiable Delay Functions. Cryptology ePrint Archive, Report 2018/712, 2018. `https://eprint.iacr.org/2018/712`.

[31] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 561–586, Cham, 2019. Springer International Publishing.

[32] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. Cryptology ePrint Archive, Report 2018/1188, 2018. `https://eprint.iacr.org/2018/1188`.

[33] Dan Boneh and Matthew Franklin. Efficient generation of shared RSA keys. In Burton S. Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, pages 425–439, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[34] Dan Boneh, Ben Lynn, and Hovav Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, Sep 2004.

[35] Joseph Bonneau. EthIKS: Using Ethereum to audit a CONIKS key transparency log. BITCOIN'16, 2016.

[36] Sean Bowe. Switch from BN254 to BLS12-381. https://github.com/zcash/zcash/issues/2502, July 2017. Accessed: 2019-02-03.

[37] 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 '18*, 2018.

[38] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[39] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[40] Elette Boyle and Rafael Pass. Limits of Extractability Assumptions with Distributional Auxiliary Input. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 236–261, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[41] J. Buchmann and H. C. Williams. A Key-exchange System Based on Imaginary Quadratic Fields. *J. Cryptol.*, 1(2):107–118, August 1988.

[42] Johannes Buchmann and Safuat Hamdy. A Survey on IQ Cryptography. In *In Proceedings of Public Key Cryptography and Computational Number Theory*, pages 1–15, 2001.

[43] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable Certificate Management using Undeniable Attestations. In *ACM CCS'00*, 2000.

[44] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE S&P'18*, 2018.

[45] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK Compilers. Cryptology ePrint Archive, Report 2019/1229, 2019. https://eprint.iacr.org/2019/1229.

[46] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology*, 18(3):219–246, Jul 2005.

[47] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and Modular Anonymous Credentials: Definitions and Practical Constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 262–288, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[48] Jan Camenisch and Markus Stadler. Proof Systems for General Statements about Discrete Logarithms. Technical report, ETH Zurich, 1997.

[49] Sébastien Canard and Aline Gouget. Multiple Denominations in E-cash with Compact Transaction Data. In Radu Sion, editor, *Financial Cryptography and Data Security*, pages 82–97, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[50] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive Security for Threshold Cryptosystems. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[51] John Canny and Stephen Sorkin. Practical Large-Scale Distributed Key Generation. In Christian Cachin and Jan L. Camenisch, editors, *Advances in Cryptology - EUROCRYPT 2004*, pages 138–152, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[52] Ignacio Cascudo and Bernardo David. SCRAPE: Scalable Randomness Attested by Public Entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security*, pages 537–556, Cham, 2017. Springer International Publishing.

[53] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *TOCS*, 20(4), 2002.

[54] Dario Catalano and Dario Fiore. Vector Commitments and Their Applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *Public-Key Cryptography – PKC 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[55] Dario Catalano, Dario Fiore, and Mariagrazia Messina. Zero-Knowledge Sets with Short Proofs. In Nigel Smart, editor, *EUROCRYPT'08*, pages 433–450, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[56] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less</> trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1639–1656, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Melissa Chase and Sarah Meiklejohn. Transparency Overlays and Applications. In *ACM CCS'16*, 2016.

[58] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation. Cryptology ePrint Archive, Report 2018/968, 2018. `https://eprint.iacr.org/2018/968`.

[59] Chia Network. BLS signatures in C++ using the RELIC toolkit. `https://github.com/Chia-Network/bls-signatures`. Accessed: 2019-05-06.

[60] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-Quantum and Transparent Recursive Proofs from Holography. Cryptology ePrint Archive, Report 2019/1076, 2019. `https://eprint.iacr.org/2019/1076`.

[61] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395, Oct 1985.

[62] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri. Efficient gossip protocols for verifying the consistency of certificate logs. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 415–423, Sep. 2015.

[63] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[64] Scott A. Crosby and Dan S. Wallach. Efficient Data Structures for Tamper-evident Logging. In *USENIX Security '09*, 2009.

[65] Scott A. Crosby and Dan S. Wallach. Authenticated Dictionaries: Real-World Costs and Trade-Offs. *ACM Transactions on Information and System Security*, 14(2), September 2011.

[66] Rasmus Dahlberg and Tobias Pulls. Verifiable Light-Weight Monitoring for Certificate Transparency Logs. In *NordSec 2018: Secure IT Systems*, 2018.

[67] Rasmus Dahlberg, Tobias Pulls, Jonathan Vestin, Toke Høiland-Jørgensen, and Andreas Kassler. Aggregation-Based Gossip for Certificate Transparency. *CoRR*, abs/1806.08817, 2018.

[68] Ivan Damgård and Eiichiro Fujisaki. An integer commitment scheme based on groups with hidden order. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002*, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[69] Ivan Damgård and Maciej Koprowski. Generic Lower Bounds for Root Extraction and Signature Schemes in General Groups. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 256–271, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[70] Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, Robust and Constant-Round Distributed RSA Key Generation. In Daniele Micciancio, editor, *Theory of Cryptography*, pages 183–200, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[71] Ivan Damgård and Nikos Triandopoulos. Supporting Non-membership Proofs with Bilinear-map Accumulators. Cryptology ePrint Archive, Report 2008/538, 2008. `https://eprint.iacr.org/2008/538`.

[72] Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 66–98, Cham, 2018. Springer International Publishing.

[73] Yvo Desmedt. Society and Group Oriented Cryptography: a New Concept. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 120–127, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[74] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 307–315, New York, NY, 1990. Springer New York.

[75] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 457–469, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[76] DFINITY. Distributed Key Generation in JS. `https://github.com/dfinity/dkg`. Accessed: 2019-05-07.

[77] DFINITY. go-dfinity-crypto. `https://github.com/dfinity/go-dfinity-crypto`. Accessed: 2019-05-06.

[78] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, Sep. 2006.

[79] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure Logging Schemes and Certificate Transparency. In *ESORICS'16*, 2016.

[80] Peter Eckersley. Iranian hackers obtain fraudulent HTTPS certificates: How close to a Web security meltdown did we get? `https://www.eff.org/deeplinks/2011/03/iranian-hackers-obtain-fraudulent-https`, 2011. Accessed: 2019-10-13.

[81] Peter Eckersley and Burns Jesse. Is the SSLiverse a Safe Place. `https://www.eff.org/files/ccc2010.pdf`, 2010.

[82] Graham Edgecombe. Compressing X.509 certificates. `https://www.grahamedgecombe.com/blog/2016/12/22/compressing-x509-certificates`, 2016. Accessed: 2018-04-12.

[83] Adam Eijdenberg, Ben Laurie, and Al Cutter. Verifiable Data Structures. `https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf`, 2016. Accessed: 2018-04-12.

[84] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.

[85] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate Transparency with Privacy. *PoPETs*, 2017(4), 2017.

[86] 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 *ACM CCS'14*, 2014.

[87] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. In *USENIX Security '12*, 2012.

[88] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *OSDI'10*, 2010.

[89] Paul Feldman. A Practical Scheme for Non-interactive Verifiable Secret Sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 427–438, Washington, DC, USA, 1987. IEEE Computer Society.

[90] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[91] Pierre-Alain Fouque and Jacques Stern. One Round Threshold Discrete-Log Key Generation without Private Channels. In Kwangjo Kim, editor, *Public Key Cryptography*, pages 300–316, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[92] Yair Frankel, Philip MacKenzie, and Moti Yung. Adaptively-Secure Distributed Public-Key Systems. In Jaroslav Nešetřil, editor, *Algorithms - ESA' 99*, pages 4–27, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[93] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast Distributed RSA Key Generation for Semi-honest and Malicious Adversaries. In *CRYPTO'18*, 2018.

[94] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113 – 3121, 2008. Applications of Algebra to Cryptography.

[95] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT'13*, 2013.

[96] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *Applied Cryptography and Network Security*, pages 156–174, Cham, 2016. Springer International Publishing.

[97] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust Threshold DSS Signatures. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 354–371, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[98] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Applications of Pedersen's Distributed Key Generation Protocol. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 373–390, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[99] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology*, 20(1):51–83, Jan 2007.

[100] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and Fast-track Multiparty Computations with Applications to Threshold Cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.

[101] Craig Gentry and Daniel Wichs. Separating Succinct Non-interactive Arguments from All Falsifiable Assumptions. In *STOC'11*, 2011.

[102] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[103] Glenn Greenwald. Edward Snowden: the whistleblower behind the NSA surveillance revelations. `https://www.theguardian.com/world/2013/jun/09/edward-snowden-nsa-whistleblower-surveillance`, 2013. Accessed: 2019-10-14.

[104] GNOSIS. Distributed Key Generation. `https://github.com/gnosis/dkg`. Accessed: 2019-05-07.

[105] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, June 2019.

[106] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[107] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. *Collision-Free Hashing from Lattice Problems*, pages 30–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[108] S Goldwasser, S Micali, and C Rackoff. The Knowledge Complexity of Interactive Proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[109] Google. HTTPS encryption on the web: Certificate transparency. `https://transparencyreport.google.com/https/certificates`, 2016. Accessed: 2018-04-12.

[110] Google. Trillian: General Transparency. `https://github.com/google/trillian`, 2016. Accessed: 2018-04-12.

[111] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT'10*, 2010.

[112] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[113] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[114] L. Harn. Group-oriented (t, n) threshold digital signature scheme and digital multisignature. *IEE Proceedings - Computers and Digital Techniques*, 141(5):307–313, Sep. 1994.

[115] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. In Orr Dunkelman, editor, *Topics in Cryptology – CT-RSA 2012*, pages 313–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[116] Heiko Stamer. Distributed Privacy Guard. `https://www.nongnu.org/dkgpg/`. Accessed: 2019-05-07.

[117] Amir Herzberg, Markus Jakobsson, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive Public Key and Signature Systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, CCS '97, pages 100–110, New York, NY, USA, 1997. ACM.

[118] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. In Don Coppersmith, editor, *Advances in Cryptology — CRYPT0' 95*, pages 339–352, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[119] Benjamin Hof and Georg Carle. Software Distribution Transparency and Auditability. *CoRR*, abs/1711.07278, 2017.

[120] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification. `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`, 2015. Accessed: 2017-11-17.

[121] Ingemar Ingemarsson and Gustavus J. Simmons. A Protocol to Set Up Shared Secret Schemes Without the Assistance of a Mutually Trusted Party. In Ivan Bjerre Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, pages 266–282, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[122] Stanisław Jarecki and Anna Lysyanskaya. Adaptively Secure Threshold Cryptography: Introducing Concurrency, Removing Erasures. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 221–242, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[123] Antoine Joux. A One Round Protocol for Tripartite Diffie–Hellman. In Wieb Bosma, editor, *Algorithmic Number Theory*, pages 385–393, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[124] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-End Integrity Protection for Web Applications. In *IEEE S&P'16*, 2016.

[125] A. Kate and I. Goldberg. Distributed Key Generation for the Internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128, June 2009.

[126] Aniket Kate. *Distributed Key Generation and Its Applications*. PhD thesis, Waterloo, Ontario, Canada, 2010.

[127] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed Key Generation in the Wild. Cryptology ePrint Archive, Report 2012/377, 2012. `https://eprint.iacr.org/2012/377`.

[128] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[129] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Polynomial commitments. Technical report, 2010.

[130] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 357–388, Cham, 2017. Springer International Publishing.

[131] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perring, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-key Validation Infrastructure. In *WWW'13*, 2013.

[132] Paul C. Kocher. On certificate revocation and validation. In *Financial Cryptography '98*, 1998.

[133] Markulf Kohlweiss and Alfredo Rial. Optimally Private Access Control. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, page 37–48, New York, NY, USA, 2013. Association for Computing Machinery.

[134] Russell W.F. Lai and Giulio Malavolta. Subvector Commitments with Application to Succinct Arguments. Cryptology ePrint Archive, Report 2018/705, 2018. `https://eprint.iacr.org/2018/705`.

[135] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3), 1982.

[136] Ben Laurie. Revocation Transparency. `https://www.links.org/files/RevocationTransparency.pdf`, 2015. Accessed: 2018-07-31.

[137] Ben Laurie, Adam Langley, and Emilia Kasper. RFC: Certificate Transparency. `http://tools.ietf.org/html/rfc6962`, 2013. Accessed: 2015-5-13.

[138] Laurie Law and Brian J. Matt. Finding Invalid Signatures in Pairing-Based Batches. In Steven D. Galbraith, editor, *Cryptography and Coding*, pages 34–53, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[139] Jiangtao Li, Ninghui Li, and Rui Xue. Universal Accumulators with Efficient Nonmembership Proofs. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 253–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[140] Jinyuan Li, Maxwell Krohn, David Mazières, Dennis Shasha, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[141] Jinyuan Li and David Maziéres. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 10–10, Berkeley, CA, USA, 2007. USENIX Association.

[142] Benoît Libert and Moti Yung. Concise Mercurial Vector Commitments and Independent Zero-Knowledge Sets with Short Proofs. In Daniele Micciancio, editor, *Theory of Cryptography*, pages 499–517, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[143] Benoît Libert, Somindu C. Ramanna, and Moti Yung. Functional Commitment Schemes: From Polynomial Commitments to Pairing-Based Accumulators from Simple Assumptions. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[144] Helger Lipmaa. Secure Accumulators from Euclidean Rings without Trusted Setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, pages 224–240, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[145] Ravi Mandalia. Security breach in CA networks - Comodo, DigiNotar, GlobalSign. `http://blog.isc2.org/isc2_blog/2012/04/test.html`, 2012. Accessed: 2015-08-22.

[146] Petros Maniatis and Mary Baker. Authenticated Append-only Skip Lists. *CoRR*, cs.CR/0302010, 2003.

[147] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. Bringing Deployable Key Transparency to End Users. In *USENIX Security '15*, August 2015.

[148] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with Assessing the Impact of NFS Advances on the Security of Pairing-Based Cryptography. In *Mycrypt'16*, 2017.

[149] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 80–89, New York, NY, USA, 1991. ACM.

[150] Ralph C. Merkle. Method of providing digital signatures, January 1982.

[151] Silvio Micali. Computationally Sound Proofs. *SIAM Journal on Computing*, 30(4), 2000.

[152] Silvio Micali, Michael Rabin, and Joe Kilian. Zero-Knowledge Sets. In *FOCS'03*, 2003.

[153] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable Random Functions. In *FOCS'99*, 1999.

[154] Mitsunari Shigeo. BLS threshold signature. `https://github.com/herumi/bls/`. Accessed: 2019-05-06.

[155] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2008. Accessed: 2017-03-08.

[156] Namecoin. Namecoin. `https://namecoin.info/`, 2011. Accessed: 2015-08-23.

[157] Moni Naor. On Cryptographic Assumptions and Challenges. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 96–109, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[158] Moni Naor and Kobbi Nissim. Certificate Revocation and Certificate Update. In *USENIX Security '98*, 1998.

[159] Wafa Neji, Kaouther Blibech, and Narjes Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Security and Communication Networks*, 9(17):4585–4595, 2016.

[160] Lan Nguyen. Accumulators from Bilinear Pairings and Applications. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 275–292, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[161] André Niemann and Jacqueline Brendel. A Survey on CA Compromises, 2014.

[162] 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 '17*, 2017.

[163] Alina Oprea and Kevin D. Bowers. Authentic Time-Stamps for Archival Storage. In *ESORICS'09*, 2009.

[164] Orbs Network. Orbs Network: DKG for BLS threshold signature scheme on the EVM using solidity. `https://github.com/orbs-network/dkg-on-evm`, 2018. Accessed: 2019-02-15.

[165] Mark H. Overmars. *Design of Dynamic Data Structures*. 1987.

[166] Mark H. Overmars and Jan van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4), 1981.

[167] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of Correct Computation. In Amit Sahai, editor, *Theory of Cryptography*, pages 222–242, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[168] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming Authenticated Data Structures. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 353–370, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[169] Charalampos Papamanthou and Roberto Tamassia. Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures. In *Information and Communications Security*, 2007.

[170] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal Verification of Operations on Dynamic Sets. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 91–110, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[171] Choonsik Park and Kaoru Kurosawa. New ElGamal Type Threshold Digital Signature Scheme. *IEICE Trans. Fundamentals, A*, 79(1):86–93, jan 1999.

[172] Torben Pryds Pedersen. A Threshold Cryptosystem without a Trusted Party. In Donald W. Davies, editor, *Advances in Cryptology – EUROCRYPT '91*, pages 522–526. Springer Berlin Heidelberg, 1991.

[173] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In Joan Feigenbaum, editor, *Proceedings on Advances in Cryptology*, CRYPTO '91, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[174] Roel Peeters and Tobias Pulls. Insynd: Improved Privacy-Preserving Transparency Logging. In *ESORICS'16*, 2016.

[175] Chris Peikert. A Decade of Lattice Cryptography. Cryptology ePrint Archive, Report 2015/939, 2015. `https://eprint.iacr.org/2015/939`.

[176] Philipp Schindler. Ethereum-based Distributed Key Generation Protocol. `https://github.com/PhilippSchindler/ethdkg`. Accessed: 2019-05-07.

[177] Krzysztof Pietrzak. Simple Verifiable Delay Functions. Cryptology ePrint Archive, Report 2018/627, 2018. https://eprint.iacr.org/2018/627.

[178] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *NSDI'14*, 2014.

[179] F. P. Preparata and D. V. Sarwate. Computational Complexity of Fourier Transforms Over Finite Fields. *Mathematics of Computation*, 31(139):740–751, 1977.

[180] Tobias Pulls and Roel Peeters. Balloon: A Forward-Secure Append-Only Persistent Authenticated Data Structure. In *ESORICS'15*, 2015.

[181] Leonid Reyzin and Sophia Yakoubov. *Efficient Asynchronous Accumulators for Distributed PKI*, pages 292–309. Springer International Publishing, Cham, Switzerland, 2016.

[182] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[183] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS'14*, February 2014.

[184] Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, Auditable Membership Proofs. In Yair Frankel, editor, *Financial Cryptography*, pages 53–71, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[185] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar Weippl. ETHDKG: Distributed Key Generation with Ethereum Smart Contracts. Cryptology ePrint Archive, Report 2019/985, 2019. https://eprint.iacr.org/2019/985.

[186] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[187] Berry Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 148–164, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[188] A. Schönhage. Schnelle Berechnung Von Kettenbruchentwicklungen. *Acta Inf.*, 1(2):139–144, June 1971.

[189] SCIPR Lab. libff. https://github.com/scipr-lab/libff, 2016. Accessed: 2018-07-28.

[190] SCIPR Lab. libfqfft. https://github.com/scipr-lab/libfqfft, 2016. Accessed: 2018-07-28.

[191] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979.

[192] Victor Shoup. Practical Threshold Signatures. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[193] Victor Shoup. libntl. `https://www.shoup.net/ntl/`, 2016. Accessed: 2018-07-28.

[194] Ryan Sleevi. Certificate Transparency in Chrome - Change to Enforcement Date. `https://groups.google.com/a/chromium.org/forum/#!msg/ct-policy/sz_3W_xKBNY/6jq2ghJXBAAJ`, 2017. Accessed: 2018-04-20.

[195] Markus Stadler. Publicly Verifiable Secret Sharing. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 190–199, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[196] Douglas R. Stinson and Reto Strobl. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In Vijay Varadharajan and Yi Mu, editors, *Information Security and Privacy*, pages 417–434, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[197] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545, May 2016.

[198] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fischer, and Bryan Ford. Scalable Bias-Resistant Distributed Randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460, 2017.

[199] Pawel Szalachowski, Stephanos Matsumoto, and Adrian Perrig. PoliCert: Secure and Flexible TLS Certificate Management. In *ACM CCS'14*, 2014.

[200] A. Tomescu and S. Devadas. Catena: Efficient Non-equivocation via Bitcoin. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 393–409, May 2017.

[201] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency Logs via Append-Only Authenticated Dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1299–1316, New York, NY, USA, 2019. Association for Computing Machinery.

[202] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards Scalable Threshold Cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020.

[203] Jelle van den Hooff, M. Frans Kaashoek, and Nickolai Zeldovich. VerSum: Verifiable Computations over Large Public Logs. In *ACM CCS'14*, 2014.

[204] VMware. threshsign library. https://github.com/vmware/concord-bft/tree/master/threshsign. Accessed: 2019-05-06.

[205] Joachim von zur Gathen and Jurgen Gerhard. Fast Euclidean Algorithm. In *Modern Computer Algebra*, chapter 11, pages 313–333. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[206] Joachim von zur Gathen and Jurgen Gerhard. Fast Multiplication. In *Modern Computer Algebra*, chapter 8, pages 221–254. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[207] Joachim von zur Gathen and Jurgen Gerhard. Fast polynomial evaluation and interpolation. In *Modern Computer Algebra*, chapter 10, pages 295–310. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[208] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In *IEEE S&P'18*, 2018.

[209] Benjamin Wesolowski. Efficient Verifiable Delay Functions. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 379–407, Cham, 2019. Springer International Publishing.

[210] Wikipedia contributors. Bit-reversal permutation — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 24-July-2019].

[211] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf, 2015. Accessed: 2016-05-15.

[212] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security '18*, 2018.

[213] Jiangshan Yu, Vincent Cheval, and Mark Ryan. DTKI: A New Formalized PKI with Verifiable Trusted Parties. *The Computer Journal*, 59(11), 2016.

[214] Zcash. What is Jubjub. https://z.cash/technology/jubjub/, 2017. Accessed: 2019-02-03.

[215] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. Cryptology ePrint Archive, Report 2019/1482, 2019. https://eprint.iacr.org/2019/1482.