Abstract of "Encrypted Distributed Storage Systems" by Archita Agarwal, Ph.D. Candidate, Brown University, May 2021.

In the era of big data, distributed data stores have become an indispensable necessity. Almost all these data stores store sensitive data such as our voicemails, messages, emails, shopping data, health records, etc. Unfortunately, with an increasing number of data breaches on a regular basis, privacy has become a major concern. Encryption is often proposed as a solution to privacy issues, however, naively applied encryption leads to brittle systems that fail to provide satisfactory privacy guarantees; it is only through careful system-wide analysis that complicated systems can achieve provable privacy.

This thesis formalizes the use of end-to-end encryption (where data is kept encrypted at all times) in distributed hash tables (DHTs) and key-value stores (KVSs). Both are fundamental in the design of storage systems we use today. For example, Amazon's Dynamo KVS underlies its shopping cart, Facebook's Cassandra KVS supports Messenger and Google's Bigtable KVS manages data in Gmail. Integrating end-to-end encryption into these basic building blocks would therefore allow us to support privacy-preserving systems which would greatly increase the confidentiality of our data.

In particular, we introduce the notion of encrypted DHTs and encrypted KVSs and provide security definitions that capture the security properties one would desire from such encrypted systems. We then isolate the key properties (such as load balancing, equivocation) needed from the plaintext DHTs and KVSs to have secure constructions. Finally, we give constructions of encrypted DHTs and encrypted KVSs and formally analyze their security under our security definitions. Also, to show that the required properties are indeed achievable in practice, we study common DHTs and KVSs and show that they satisfy all the requirements.

Encrypted Distributed Storage Systems

by
Archita Agarwal

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2021

This dissertation by Archita Agarwal is accepted in its present form by
the Department of Computer Science as partially satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date 3/11/21

Seny Kamara, Advisor
Brown University

Recommended to the Graduate Council

Date 3/18/21

Ittai Abraham, Reader
VMware Research

17 March 2021

Date

Maurice Herlihy, Reader
Brown University

Date 3/20/2021

Malte Schwarzkopf, Reader
Brown University

Approved by the Graduate Council

Date

Andrew G. Campbell
Dean of the Graduate School

Acknowledgements

This dissertation is the result of seven years of support from amazing people. First and foremost, I would like to thank my advisor, Seny Kamara, for his tremendous help and guidance throughout my doctoral research. Seny has been a truly supportive advisor, an inspiring mentor and a great role model. He has taught me how to be a passionate researcher as well as a tolerant and effective teacher. His advice has helped me to develop both as a researcher and as a person. I cannot describe how blessed I feel to have him as my advisor.

I am very grateful to my committee member, Maurice Herlihy, with whom I had the pleasure of collaborating with in the earlier half of my Ph.D. He taught me to have a broad perspective on things and not get bogged down by the tiny details. Unknowingly, he has also taught me so much about presentation skills, an invaluable skill for a person who aspires to be an educator. I also had the wonderful opportunity of collaborating with Malte Shwarzkopf, my other committee member, who has always been welcoming and accommodating during our meetings to discuss any problem I had. My heartfelt thanks to Shriram Krishnamurthi for his insightful career advice and support. I might have dropped out of the program had he not encouraged me during the times when I was unsure of what to do.

I also feel extremely lucky to have Tarik Moataz as my mentor. He has been a source of unwavering support throughout these years. He has always made himself available for me, be it for research meetings or for personal discussions. There have been uncountable instances when I went to his office as a nutcase but left as a sorted one. It has been an absolute privilege to know him!

My days in Providence would not have been the same without the friends I made in the department and outside of it and I am grateful to each one of them. To Sagar Wadhwa, for his company in all of our PhD adventures. His companionship has forever made me a better person. To Cyrus Cousins, for all those long brainstorming sessions on probability theory; you are truly a genius. To Anshul Jain, Kaushik Vijaykumar, Jay Seth, Karthik Desingh, Siddhant Jaitpal for making Providence home for me; I thoroughly enjoyed our weekly get togethers for movies, dinners, cricket matches, general chatting, just to name a few. To Apoorvaa Deshpande, Pinkesh Malhotra and Sagar for being wonderful friends; starting my PhD with you three and going through all the milestones together definitely made the journey more fun. I will also never forget all the rock climbing, hiking, camping and friday night outs with Ravi Kumar, Alexandra Papoutsaki, Dan Berg and Kaushik; in some way you guys taught me not to give up! I am also thankful to Marilyn George,

my roommate for three years and my academic sibling, for the numerous conversations over the years (and also for fixing my writeups). Huge thanks to my GCB buddies, Ghous Amjad, Nediyana Daskalova, Vikram Saraph, Daniel Engel and Marilyn for keeping PhD fun. There are many others I would like to thank for all the adventures during my time in Providence, though the list is just too long to list here.

Last but not least, my deepest gratitude goes to my parents for always keeping me on the right path and making me stronger to dream for the best. Without them, I could not have undertaken this journey, far away from home. Thanks to my mother and father for showing faith and belief in me, no matter what. To my sisters and brothers-in-law for supporting me in all my good and bad times. None of this would have been possible without you! Lots of love.

Contents

Chapter 1

Introduction

In the era of big data, distributed data stores (DDS) have become an indispensable necessity. The huge benefit of using these DDSs comes in terms of efficiency, availability and scalability. Almost all the DDSs store sensitive data about us such as our voicemails, messages, emails, shopping data, health records, etc. Unfortunately, with an increasing number of data breaches on a regular basis, privacy has become a major concern. Moreover, due to their distributed nature, they are harder to secure than their non-distributed counterparts.

Encryption is often proposed as a solution to some of the privacy concerns, however, naively applied encryption leads to brittle systems that might fail to provide satisfactory privacy guarantees. For instance, encrypting data at rest and decrypting it before use might not be enough because each decryption exposes the data and increases its likelihood of being stolen. Therefore, to overcome this problem, cryptographers have started developing end-to-end encrypted solutions. End-to-end encryption ensures that data is kept encrypted, *even in use*, and is therefore one of the best ways to achieve data confidentiality.

In this thesis, I initiate the study of end-to-end encryption in DDSs. In particular, I design and analyze schemes for storing, querying and updating encrypted data in data stores. I study two kinds of data stores, first store data in distributed hash tables (DHTs), and the second on blockchains. DHT based data stores are extremely popular and have applications in content delivery networks, P2P networks, key-value stores, distributed file systems, just to name a few. On the other hand, blockchain based data stores are slowly gaining popularity as they try to provide guarantees of both blockchains and traditional databases, like decentralization, tamper-proofness and support for complex queries. Integrating end-to-end encryption into these building blocks would therefore allow us to support complex privacy-preserving systems that are built on top of them and would greatly enhance the confidentiality of our data.

1.1 Thesis at a Glance

Before giving an overview of individual chapters in this thesis, I first describe the two main themes that run through all my works.

- **Formalizing systems**: I used the methodology of *provable security* to formally study the security of end-to-end encrypted DDSs (called encrypted DDSs henceforth). Under this framework, each encrypted DDS is analyzed with respect to a specific *security definition* and against a particular threat model. There are two steps, first formally defines the encrypted DDS and the next its security. Formalizing the encrypted DDS is a non-trivial step as it requires abstraction of the core components of the underlying DDS, such as, how the DDS nodes store data, how they talk to each other, what happens in the event of node failures, to name a few. The second step requires designing a security definition that captures the security properties one would desire from an encrypted DDS. Since a system is only as secure as its security definition, it is crucial that the definition accurately models the threats in the real world.

- **Designing and analyzing systems**: I also designed encrypted DDSs and analyzed their security under formal security definitions. The analysis involved exploring fundamental connections between privacy and distributed systems. For instance, a key question in the theory of distributed systems is the connection between a machine's *local* knowledge and its *global* knowledge. Privacy is also concerned about the same question – how much information does a corrupted machine have about the global system?

Together, my work tries to answer how practical distributed data stores and encryption interact and tries to understand the ways in which they leak information. My thesis is divided into three chapters as follows.

1.1.1 Encrypted Distributed Hash Tables

Distributed Hash Tables. Distributed hash tables (DHTs) are the most fundamental building block in the design of highly scalable and reliable systems. DHTs are decentralized and distributed systems that store label/value pairs (ℓ, v) and support *get* and *put* operations. The former is used to store pairs in DHTs while the latter is used to retrieve pairs from DHTs. DHTs are distributed in the sense that the pairs are stored across multiple nodes instead of a single node. DHTs provide many useful properties but the most important are *load balancing* and *fast data retrieval and storage* even in highly-transient networks (i.e., where storage nodes join and leave at high rates). It is hard to overstate the impact that the DHTs have had on systems design. For example, DHTs enable Amazon to handle over a billion purchases a year and Facebook to support two billion users worldwide.

Results. In Chapter 3, we develop a framework that allows us to formally analyze encryption in context of DHTs. We isolate two properties of DHTs, *balance* and *non-committing allocation*, and show that the security of *any* end-to-end encrypted DHT is a function of these two properties. To

show that the balance and non-committing allocations are indeed achievable in practice, we study Chord [78]—which is arguably the most influential DHT– and show that it satisfies both these properties. This work is under submission [24].

Insights. The work challenged our initial intuitions in many ways and helped us gain a deeper understanding of the intricate relationships between privacy and distributed systems. For example, suppose a subset of nodes are corrupted and collude. During the operation of this DHT, what information can the corrupted nodes learn about our data? A-priori, it might seem that the only information they can learn is related to what they collectively hold (i.e., the union of the data they store). For example, they might learn that there are at least m pairs stored in the DHT, where m is the sum of the number of pairs held by each corrupted node. While this intuition might seem correct, it is not true! In fact, the corrupted nodes can infer additional information about data they do not hold. For example, they can infer a good approximation on the *total* number of pairs in the system even if they collectively hold a small fraction of it. Here, the problem is that since DHTs are *load balanced*, with high probability, each node receives approximately the same number of pairs. Because of this, the corrupted nodes can guess that, with high probability, the total number of pairs in the system is about mn/t, where t is the number of corrupted nodes and n is the total number of nodes. While this may seem benign, this is just one example to highlight that finding and analyzing information leakage in distributed systems can be non-trivial. In fact, otherwise desirable properties of distributed systems (e.g., load balancing) can have subtle effects on security.

1.1.2 Encrypted Key-Value Stores

Key-Value Stores. Like DHTs, key-value stores (KVSs) store label/value pairs which can be accessed by *get* and *put* operations. Unlike DHTs, they replicate the same pair on *multiple* nodes instead of storing it on a single one. KVSs are fault-tolerant because a value can be accessed even if some nodes fail. KVSs provide a simple data model but have become fundamental to modern systems due to their high performance, scalability and availability. For example, Amazon's Dynamo [55] KVS underlies its shopping cart, Facebook's Cassandra [82] KVS supports Messenger and Google's Bigtable [51] KVS manages data in Gmail.

Results. In Chapter 4, we extend our previous framework to encrypted KVSs and show that the encrypted KVSs achieve the same level of security as their counterparts in non-distributed setting [52, 50]. However, we also show that if the underlying KVS satisfies *read-your-writes consistency*, then the security can be improved—effectively showing that a certain level of consistency can improve the security of a system. This work was published in Indocrypt [23].

Insights. The most challenging (and fascinating) part of this work was discovering the effect of consistency guarantees on security. Since most practical KVSs are *eventually consistent*, we wanted to formalize eventual consistency under the provable security framework. However, it turned out that

the framework was not amenable to this notion of consistency. This raised the question if eventually consistent systems are fundamentally "insecure" or just that the consistency models do not play well with the cryptographic models? We later realised that certain alternate notions of consistency did lend themselves to be formalized precisely in cryptographic settings. In particular, if the KVS is read-your-writes consistent, which is a stronger consistency notion than eventual consistency, then the encrypted KVS based on it can be shown to be secure. The precise relationship between consistency and leakage is intriguing and is still unexplored and is left for future work.

1.1.3 Encrypted Blockchain Dababases

Blockchain Databases. *Blockchain databases* are storage systems that combine properties of both blockchains and traditional databases like decentralization, tamper-proofness, low query latency and support for complex queries. As they gain wider adoption, concerns over the confidentiality of the data they manage will increase. Already, several projects [89, 14, 94] use blockchains to store sensitive data like electronic healthcare and financial records, legal documents, and customer data.

Results. In Chapter 5, we develope ways of storing modifiable encrypted inverted-indices on top of blockchains. All our schemes are legacy-friendly in the sense that they could be used with any blockchain. We also implement our schemes on Algorand [1] and Ethereum [5] blockchains and evaluated their efficiency emperically. This work was published in AFT [22].

Insights. The work has several challenges, the biggest being that blockchains are tamper-proof and yet we need a way to update the inverted-indices stored on top of them. A simple idea is to write an entirely new index every time an update is made. Unfortunately, in case of blockchains, every write costs money making this a very expensive solution. Moreover, in blockchains, writes are not just expensive, but also are very slow. For example, in Bitcoin, a transaction (a.k.a a write) needs to become 6 blocks deep to be considered "stable". Therefore, we design new techniques that took these factors into account and help us achieve efficiency — money-wise, time-wise and storage-wise — both for reads and writes.

At a high-level, to reduce our write-complexity, we model the inverted-index in such a way that it only has a few dependencies between its sub-components and then we write all the independent sub-components in parallel to the blockchain. All the components sent to the blockchain at the same time, "stabilize" almost together, saving us time from having to wait for each one of them to stabilize individually.

Chapter 2

Preliminaries

Notation. The set of all binary strings of length n is denoted as $\{0,1\}^n$, and the set of all finite binary strings as $\{0,1\}^*$. $[n]$ is the set of integers $\{1,\ldots,n\}$, and $2^{[n]}$ is the corresponding power set. We write $x \leftarrow \chi$ to represent an element x being sampled from a distribution χ, and $x \xleftarrow{\$} X$ to represent an element x being sampled uniformly at random from a set X. The output x of an algorithm $\mathcal{A}$ is denoted by $x \leftarrow \mathcal{A}$. Given a sequence $\mathbf{v}$ of n elements, we refer to its i^{th} element as v_i or $\mathbf{v}[i]$. If S is a set then $|S|$ refers to its cardinality. If s is a string then $|s|_2$ refers to its bit length. We denote by $\mathsf{Ber}(p)$ the Bernoulli distribution with parameter p.

Dictionaries. A dictionary structure DX of capacity n holds a collection of n label/value pairs $\{(\ell_i, v_i)\}_{i\leq n}$ and supports get and put operations. We write $v_i := \mathsf{DX}[\ell_i]$ to denote getting the value associated with label ℓ_i and $\mathsf{DX}[\ell_i] := v_i$ to denote the operation of associating the value v_i in DX with label ℓ_i.

Multi-maps. A multi-map structure MM with capacity n is a collection of n label/tuple pairs $\{(\ell_i, \mathbf{v}_i)\}_{i\leq n}$ that supports get and put operations. Similar to dictionaries, we write $\mathbf{v}_i := \mathsf{MM}[\ell_i]$ to denote getting the tuple associated with label ℓ_i and $\mathsf{MM}[\ell_i] := \mathbf{v}_i$ to denote operation of associating the tuple $\mathbf{v}_i$ to label ℓ_i.

2.1 Cryptographic Vocabulary

Views. The view of a node N that participates in the execution of a randomized experiment **Exp** consists of its random coins and all messages that it sends and receives. This is a random variable which we denote by $\mathsf{view}_{\mathbf{Exp}}(N)$. When the experiment is clear from context we omit the subscript for visual clarity. We sometimes consider the joint random variable consisting of the views of multiple nodes. If S is a set of nodes, we denote by $\mathsf{view}_{\mathbf{Exp}}(S)$ the joint random variable $\langle \mathsf{view}_{\mathbf{Exp}}(N)\rangle_{N\in S}$.

The random oracle model. The random oracle model is a paradigm in which a cryptographic protocol is proven secure assuming oracle access to a random function which is instantiated in practice with a cryptographic hash function. Access to random oracles is a powerful assumption that enables cryptographers to design protocols that achieve strong security properties.

Leakage profiles. Many cryptographic primitives and protocols leak information. Examples include encryption schemes, which reveal the length of the plaintext; secure multi-party computation protocols, which (necessarily) reveal about the parties' inputs whatever can be inferred from the output(s); order-preserving encryption schemes, which reveal implicit and explicit bits of the plaintext; structured encryption schemes which reveal correlations between queries; and oblivious algorithms which reveal their runtime and the volume of data they read. Leakage-parameterized security definitions [53, 52] extend the standard provable security paradigm used in cryptography by providing adversaries (and simulators) access to leakage over plaintext data. This leakage is formally and precisely captured by a leakage profile which can then be analyzed through cryptanalysis and further theoretical study.

2.2 Cryptographic Primitives

Encryption Scheme. A private-key encryption scheme is a set of three polynomial-time algorithms $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that Gen is a probabilistic algorithm that takes a security parameter k and returns a secret key K; Enc is a probabilistic algorithm takes a key K and a message m and returns a ciphertext c; Dec is a deterministic algorithm that takes a key K and a ciphertext c and returns m if K was the key under which c was produced. Informally, a private-key encryption scheme is secure against chosen-plaintext attacks (CPA) if the ciphertexts it outputs do not reveal any partial information about the plaintext even to an adversary that can adaptively query an encryption oracle. We say a scheme is random-ciphertext-secure against chosen-plaintext attacks (RCPA) if the ciphertexts it outputs are computationally indistinguishable from random even to an adversary that can adaptively query an encryption oracle.[1]

Pseudo random functions. A family of functions $F_K : \{0,1\}^m \to \{0,1\}^n$, indexed by a key $K \in \{0,1\}^k$ is said to be *pseudorandom* if the value $F_K(x)$ is efficiently computable given K and x and the outputs of the function F_K cannot be efficiently distinguished from a uniformly chosen random function $R : \{0,1\}^m \to \{0,1\}^n$.

2.2.1 Structured Encryption

Structured encryption (STE) schemes [52] encrypt data structures in such a way that they can support operations on encrypted data. STE schemes can be distinguished depending on the type

[1] RCPA-secure encryption can be instantiated practically using either the standard PRF-based private-key encryption scheme or, e.g., AES in counter mode.

of operations they support. This includes *non-interactive* and *interactive* schemes where the former require only a single message while the latter require several rounds for queries and updates. STE schemes can also be *static* or *dynamic* where the former do not support update operations whereas the latter do. STE schemes can also be *response-revealing* or *response-hiding*, where the former reveal the response to queries whereas the latter do not. We formally define an interactive dynamic response-hiding STE as follows:

Definition 2.2.1 (Dynamic response-hiding STE)**.** *A dynamic response hiding STE scheme* $\Sigma_{\mathsf{EDS}} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Resolve}, \mathsf{Edit}^+, \mathsf{Edit}^-)$ *consists of four protocols that work as follows:*

- $(st, K; \mathsf{EDS}) \leftarrow \mathsf{Init}_{\mathbf{C},\mathbf{S}}(1^k; \perp)$ *is a probabilistic protocol between the client* $\mathbf{C}$ *and the server* $\mathbf{S}$*. The client inputs the security parameter* k *while the server inputs nothing. The client receives an empty state* st*, a key* K*, while the server receives an empty encrypted data structure* EDS*.*
- $(st', r; \perp) \leftarrow \mathsf{Query}_{\mathbf{C},\mathbf{S}}(st, K, \ell; \mathsf{EDS})$ *is a (probabilistic) protocol between the client* $\mathbf{C}$ *and the server* $\mathbf{S}$*. The client inputs the state* st*, the key* K *and the label* ℓ*, while the server inputs the encrypted data structure* EDS*. The client receives a response* r *and the server receives nothing.*
- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Edit}^+{}_{\mathbf{C},\mathbf{S}}(st, K, \ell, \mathbf{v}; \mathsf{EDS})$*: is a (probabilistic) protocol between the client* $\mathbf{C}$ *and the server* $\mathbf{S}$*. The client inputs its state* st*, the key* K*, a label* ℓ*, and a value tuple* $\mathbf{v}$*, while the server inputs the encrypted data structure* EDS*. As output, the client receives an updated state* st' *and the server receives an updated state* EDS'*.*
- $(st'; \mathsf{EDS}') \leftarrow \mathsf{Edit}^-{}_{\mathbf{C},\mathbf{S}}(st, K, \ell, \mathbf{v}; \mathsf{EDS})$*: is a (probabilistic) protocol between the client* $\mathbf{C}$ *and the server* $\mathbf{S}$*. The client inputs its state* st*, the key* K*, a label* ℓ *and a value tuple* $\mathbf{v}$*, while the server inputs the encrypted data structure* EDS*. As output, the client receives an updated state* st' *and the server receives an updated state* EDS'*.*

We say that a dynamic response hiding STE scheme Σ_{EDS} *is correct if for all* $k \in \mathbf{N}$*, for all* $(st_0, K, \mathsf{EDS}_0)$ *output by* $\mathsf{Init}(1^k; \perp)$*, for all sequences of* $m = \mathsf{poly}(k)$ *operations* $\mathsf{op}_1, \ldots, \mathsf{op}_m$*, for all* $i \in [m]$*, if* op_i *is a query* q_i*,* $\mathsf{Query}(st_{i-1}, K, \ell; \mathsf{EDS}_{i-1})$ *returns the correct response with all but negligible probability; where* st_{i-1} *is the output of the* Edit^+*,* Edit^- *or* Query *protocols, while* EDS_{i-1} *is either the output of the last update* $\rho < i$ *if it exists, or the output of the* Init *protocol otherwise.*

Security. The standard notion of security for STE guarantees that: (1) an encrypted data structure reveals no information about its underlying data structure beyond the init leakage $\mathcal{L}_{\mathsf{I}}$; (2) that the query protocol reveals no information about the data structure and the queries beyond the query leakage $\mathcal{L}_{\mathsf{Q}}$; and that (3) the Edit^+ and Edit^- protocols reveal no information about the data structure and the updates beyond the edit leakage $\mathcal{L}_{\mathsf{E}^+}/\mathcal{L}_{\mathsf{E}^-}$. If this holds for adaptively chosen operations then the scheme is said to be adaptively secure.

Definition 2.2.2 (Adaptive security of STE [53, 52])**.** *Let* $\Sigma_{\mathsf{EDS}} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Resolve}, \mathsf{Edit}^+, \mathsf{Edit}^-)$ *be a dynamic STE scheme and consider the following probabilistic experiments where* $\mathcal{A}$ *is a stateful adversary,* Sim *is a stateful simulator,* $\mathcal{L}_{\mathsf{I}}$*,* $\mathcal{L}_{\mathsf{Q}}$*,* $\mathcal{L}_{\mathsf{E}^+}$ *and* $\mathcal{L}_{\mathsf{E}^-}$ *are leakage profiles and* $z \in \{0,1\}^*$*:*

$\mathbf{Real}_{\Sigma,\mathcal{A}}(k)$: *given z the adversary $\mathcal{A}$ receives an empty encrypted data structure* EDS *from the challenger, where* $(st, K; \mathsf{EDS}) \leftarrow \mathsf{Init}(1^k; \perp)$. *The adversary then* adaptively *chooses a polynomial number of operations* $\mathsf{op}_1, \ldots, \mathsf{op}_m$ *such that* op_i *is either a query or an update. For all* $i \in [m]$, *if* op_i *is a query* $q_i = \ell$, *the adversary and the challenger execute the protocol* Query, *and the challenger receives an updated state* st' *and a response* e *while the adversary receives nothing, where* $(st', e; \perp) \leftarrow \mathsf{Query}(st, K, \ell; \mathsf{EDS})$. *If* op_i *is an update of the form* $u_i = (\mathsf{Edit}^+, \ell, \mathbf{v})$, *the adversary and the challenger execute the protocol* Edit^+, *and the challenger receives an updated state* st', *while the adversary receives an updated encrypted data structure* EDS', *where* $(st'; \mathsf{EDS}') \leftarrow \mathsf{Edit}^+(st, K, \ell, \mathbf{v}; \mathsf{EDS})$. *On the other hand, if* op_i *is an update of the form* $u_i = (\mathsf{Edit}^-, \ell, \mathbf{v})$, *the adversary and the challenger execute the protocol* Edit^-, *and the challenger receives an updated state* st', *while the adversary receives an updated encrypted data structure* EDS', *where* $(st'; \mathsf{EDS}') \leftarrow \mathsf{Edit}^-(st, K, \ell; \mathsf{EDS})$. *Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

$\mathbf{Ideal}_{\Sigma,\mathcal{A},\mathsf{Sim}}(k)$: *given* z *and leakage* $\mathcal{L}_{\mathsf{I}}(\mathsf{DS})$ *(where* $\mathsf{DS} = \perp$*) from the challenger, the simulator* Sim *returns an empty encrypted data structure* EDS *to* $\mathcal{A}$. *The adversary then* adaptively *chooses a polynomial number of operations* $\mathsf{op}_1, \ldots, \mathsf{op}_m$ *such that* op_i *is either a query or an update. For all* $i \in [m]$, *if* $\mathsf{op}_i = \ell$, *the simulator receives the query leakage* $\mathcal{L}_{\mathsf{Q}}(\mathsf{DS}, \ell)$ *and executes* Query *with the adversary. The adversary receives nothing as output. If* $\mathsf{op}_i = (\mathsf{Edit}^+, \ell, \mathbf{v})$, *the simulator receives the add leakage* $\mathcal{L}_{\mathsf{E}^+}(\mathsf{DS}, \ell, \mathbf{v})$ *and executes* Edit^+ *with the adversary. The adversary receives an updated encrypted data structure* EDS' *as output. If, on the other hand,* $\mathsf{op}_i = (\mathsf{Edit}^-, \ell, \mathbf{v})$, *the simulator receives the delete leakage* $\mathcal{L}_{\mathsf{E}^-}(\mathsf{DS}, \ell, \mathbf{v})$ *and executes* Edit^- *with the adversary. The adversary receives an updated encrypted data structure* EDS' *as output. Finally,* $\mathcal{A}$ *outputs a bit* b *that is output by the experiment.*

We say that Σ_{EDS} *is adaptively* $(\mathcal{L}_{\mathsf{I}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}^+}, \mathcal{L}_{\mathsf{E}^-})$*-secure if there exists a* PPT *simulator* Sim *such that for all* PPT *adversaries* $\mathcal{A}$, *for all* $z \in \{0,1\}^*$,

$$\left|\Pr\left[\mathbf{Real}_{\Sigma_{\mathsf{EDS}},\mathcal{A}}(k) = 1\right] - \Pr\left[\mathbf{Ideal}_{\Sigma_{\mathsf{EDS}},\mathcal{A},\mathsf{Sim}}(k) = 1\right]\right| \leq \mathsf{negl}(k).$$

Chapter 3

Encrypted Distributed Hash Tables

3.1 Introduction

In the early 2000's, the field of distributed systems was revolutionized in large part by the performance and scalability requirements of large Internet companies like Akamai, Amazon, Google and Facebook. The operational requirements of these companies—which include running services at Internet scale using commodity hardware in data centers distributed across the world—motivated the design of highly influential systems like Chord [110], Dynamo [55] and BigTable [51]. These advances in distributed systems are what enable companies like Amazon to handle over a billion purchases a year and Facebook to support two billion users worldwide.

Distributed hash tables. The most fundamental building block in the design of highly scalable and reliable systems are *distributed hash tables* (DHT). DHTs are decentralized and distributed systems that store data items associated to a label. Roughly speaking, a DHT is a distributed dictionary data structure that stores label/value pairs (ℓ, v) and that supports get and put operations. The former takes as input a label ℓ and returns the associated value v. The latter takes as input a pair (ℓ, v) and stores it. DHTs are distributed in the sense that the pairs are stored by a set of n nodes $N_1, \ldots, N_n$. To communicate and route messages to and from nodes, DHTs rely on a (usually) randomly generated *overlay network* which, intuitively, maps node names to addresses and of a distributed routing protocol that routes messages between addresses. DHTs provide many useful properties but the most important are load balancing and fast data retrieval and storage even in highly-transient networks (i.e., where storage nodes join and leave at high rates).

Classic applications of DHTs. It is hard to overstate the impact that DHTs have had on system design and listing all their possible applications is not feasible so we will recall just a few. One of the first applications of DHTs was to the design of content distribution networks (CDNs). In 1997, Karger et al. introduced the notion of consistent hashing [78] which was adopted as a core component of Akamai's CDN. Since then, many academic and industry CDNs have used DHTs for

fast content delivery [63, 109]. DHTs are also used by many P2P systems like BitTorrent [3] and its many trackerless clients including Vuze, rTorrent, Ktorrent and μTorrent. Many distributed file systems are built on top of DHTs, including CFS [54], Ivy [92], Pond [98], PAST [58].DHTs are also the main component of distributed key-value stores like Amazon's Dynamo [55] which underlies the Amazon cart, LinkedIn's Voldemort [111] and Riak [113]. Finally, many wide column NoSQL databases like Facebook's Cassandra [82], Google's BigTable [51] and Amazon's DynamoDB [106] make use of DHTs.

Off-chain storage. Currently, the field of distributed systems is going through another revolution brought about by the introduction of blockchains [93]. Roughly speaking, blockchains are distributed and decentralized storage networks with integrity and probabilistic eventual consistency. Blockchains have many interesting properties and have fueled an unprecedented amount of interest in distributed systems and cryptography. For all their appeal, blockchains have several shortcomings; the most important of which are limited storage capacity and lack of confidentiality. To address this, a lot of effort in recent years has turned to the design of distributed and/or decentralized *off-chain* storage networks whose primary purpose is to store large amounts of data while supporting fast retrieval and storage in highly transient networks. In fact, many influential blockchain projects, including Ethereum [118, 8], Enigma [122], Storj [101] and Filecoin [81] rely on off-chain storage: Ethereum, Enigma and Storj on their own custom networks and Filecoin on IPFS [34]. Due to the storage and scalability requirements of these blockchains, these off-chain storage networks often use DHTs as a core building block.

DHTs and end-to-end encryption. As discussed, DHTs are a fundamental building block in distributed systems with applications ranging from CDNs to blockchains. DHTs were originally designed for applications that mostly dealt with public data: for example, web caching or P2P file sharing. The more recent applications of DHTs, however, also need to handle *private* data. This is the case, for example, for off-chain storage networks, many of which aim to support decentralized apps for medical records, IoT data, tax information, customer records and insurance data, just to name a few. Indeed, most of these networks (e.g., Ethereum's Swarm, IPFS, Storj and Enigma) explicitly implement some form of end-to-end encryption.

The specific designs are varied but, as far as we know, none of them have been formally analyzed. This is not surprising, however, since the problem of end-to-end encryption in the context of DHTs has never been properly studied. In this work, we address this by formalizing the goals of encryption in DHTs. In particular, we introduce the notion of an *encrypted DHT* (EDHT) and propose formal syntax and security definitions for these objects. Due to the ubiquity of DHTs and the recent interest in using them to store sensitive data, we believe that a formal study of confidentiality in DHTs is a well-motivated problem of practical importance.

The standard scheme. The simplest approach to storing sensitive data on a DHT—and the one we will study in this work—is to store a label/value pair (ℓ, v) as $(F_{K_1}(\ell), \mathsf{Enc}_{K_2}(v))$ on a

standard DHT. Here, F is a pseudo-random function and Enc is a symmetric-key encryption scheme. Throughout we will refer to this as the *standard scheme*. The underlying DHT will then assign this pair to a storage node in a load balanced manner, handle routing and will move pairs around the network if a node leaves or joins. This scheme is simple and easy to implement and is, roughly speaking, what most systems implement. Ethereum's Swarm, for example, stores pairs as $\big(H(\mathrm{ct}), \mathrm{ct}\big)$, where $\mathrm{ct} \leftarrow \mathsf{Enc}_K(v)$ and H is a hash function. But is this secure? Answering this question is not simple as it is not even clear what we mean by security. But even if we were equipped with a meaningful notion of security, we will see that the answer is not straightforward. The reason is because, as we will see, the security of the standard scheme is tightly coupled with how the the underlying DHT is designed.

Information leakage in EDHTs. To illustrate this point, suppose a subset of nodes are corrupted and collude. During the operation of this DHT, what information can they learn about a client's data and/or queries? A-priori, it might seem that the only information they can learn is related to what they collectively hold (i.e., the union of the data they store). For example, they might learn that there are at least m pairs stored in the DHT, where m is the sum of the number of pairs held by each corrupted node. With respect to the client's queries they might learn, for any label handled by a corrupted node, when a query repeats. While this intuition might seem correct, it is not true. In fact, the corrupted nodes can infer additional information about data they do not hold. For example, they can infer a good approximation on the *total* number of pairs in the system even if they collectively hold a small fraction of it. Here, the problem is that DHTs are load balanced in the sense that, with high probability, each node will receive approximately the same number of pairs. Because of this, the corrupted nodes can guess that, with high probability, the total number of pairs in the system is about mn/t, where t is the number of corrupted nodes and n is the total number of nodes.

While this may seem benign, this is just one example to highlight the fact that finding and analyzing information leakage in distributed systems can be non-trivial. In fact, some of the very properties which we aim for in the context of distributed systems (e.g., load balancing) can have subtle effects on security.

3.1.1 Our Contributions

In this work, we aim to formalize the use of end-to-end encryption in DHTs and the many systems they support. As an increasing number of applications wish to store sensitive data on DHT-based systems, the use of end-to-end encryption in DHTs should be raised from a technique to a cryptographic primitive with formal syntax and security definitions. Equipped with these definitions, our goal will be to understand and study the security guarantees of the simple EDHT described above. As we will see, analyzing and proving the security of even this simple scheme is complex enough. We make several contributions.

Security of EDHTs. Our first contribution is a simulation-based definition of security for EDHTs. The definition is in the real/ideal-world paradigm commonly used to formalize the security of multi-party computation [45]. Formulating security in this way allows for definitions that are modular and intuitive. Furthermore, this seems to be a natural way to define security since DHTs are distributed objects. In our definition, we compare a real-world execution between n nodes, an honest client and an adversary, where the latter can corrupt a subset of the nodes. Roughly speaking, we say that an EDHT is secure if this experiment is indistinguishable from an ideal-world execution between the nodes, the honest client, an ideal adversary (i.e., a simulator) and an functionality that captures the ideal security properties of EDHTs. As discussed above, for any EDHT scheme, including the standard construction, there can be subtle ways in which some information about the dataset is leaked (e.g., its total size). To formally capture this, we parameterize our definition with (stateful) leakage functions that capture exactly what is or is not being revealed to the adversary. We note that our definitions handle static corruptions and are in the standalone setting.

EDHTs and structured encryption. The notion of an EDHT can be viewed and understood from the perspective of structured encryption (STE). STE schemes are encryption schemes that encrypt data structures in such a way that they can be privately queried. From this perspective, EDHTs are a form of distributed encrypted dictionaries and, in fact, one recovers the latter from the former when the network consists of only one node. We note that this connection is not just syntactical, but also holds with respect to the security definitions of both objects and to their leakage profiles. Indeed the standard scheme's leakage profile on a single-node network reduces to the leakage profile of common dictionary encryption schemes [52, 50]. This leakage, however, represents the "worst-case" leakage of the standard EDHT. This suggests that distributed STE schemes can leak less than non-distributed STE schemes which makes sense intuitively since, in the distributed setting, the adversary can only corrupt a subset of the nodes whereas in the non-distributed setting the adversary corrupts the only existing node and, therefore, all the nodes.

With this in mind, one can view our results as another approach to the recent efforts to suppress the leakage of STE schemes [76, 73]. That is, instead of (or in addition to) compiling STE schemes as in [76] or of transforming the underlying data structures as in [73], one could *distribute* the encrypted data structure.

Probabilistic leakage. Our security definition allows us to formally study any leakage produced by EDHT schemes. Interestingly, our analysis of the standard scheme will show that it achieves a very novel kind of leakage profile. Now, this leakage profile is itself quite interesting. First, it is *probabilistic* in the sense that it leaks only with some probability $p \leq 1$. As far as we know, this is the fist time such a leakage profile has been encountered. Here, the information it leaks (when it does leak) is the query equality pattern (see [76] for a discussion of various leakage patterns) which reveals if and when a query was made in the past. This is not surprising as labels are passed as $F_K(\ell)$ to the underlying DHT, which are deterministic. This leakage profile is also interesting because the probability p with which it leaks is determined by properties of the underlying DHT

and, in particular, to its load balancing properties. Specifically, the better the DHT load balances its data the smaller the probability that the EDHT will leak the query equality.

Worst-case vs. expected leakage. A-priori one might think that the adversary should only learn information related to pairs that are stored on corrupted nodes and that, since DHTs are load balanced, the total number of pairs visible to the adversary will be roughly mt/n. But there is a slight technical problem with this intuition: a DHT' s allocation of labels depends on its overlay and, for any set of corrupted nodes, there are many overlays that can induce an allocation where, say, a very large fraction of labels are mapped to corrupted nodes. The problem then is that, in the *worst-case*, the adversary could see *all* the (encrypted) pairs. We will show, however, that the intuition above is still correct because the worst-case is unlikely to occur. More precisely, we show that with probability at least $1-\delta$ over the choice of overlay, the standard scheme achieves a certain leakage profile $\mathcal{L}$ which is a function of δ (and other parameters). As far as we know, this is the first example of a leakage analysis that is not worst-case but that, instead, considers the expected leakage (with high probability) of a construction. We believe this new kind of leakage analysis is of independent interest and that the idea of expected leakage may be a fruitful direction in the design of low- or even zero-leakage schemes.

Formalizing DHTs. To better understand EDHTs and their security properties, we aim for a modular treatment. In particular, we want to isolate the properties of the underlying DHTs that have an effect on security and decouple the components of the system that have to do with the DHT from the cryptographic primitives we use like encryption and PRFs. This is in line with how systems designers use encryption in DHTs; as far as we know, all DHT-based systems that support end-to-end encryption add encryption on top of an "unmodified" DHT. Our first step, therefore, is to formally define DHTs. This includes a formal syntax but, more interestingly, a useful abstraction of the core components of a DHT including, their network overlays, their allocations (i.e., how they map label/value pairs to nodes) and their routing components.

Properties of DHTs. As mentioned above, we found that the security of the standard EDHT scheme is tightly coupled with two main properties of DHTs. More precisely, we discovered that the former's leakage is affected by a property we call *balance* which, roughly speaking, means that with probability at least $1-\delta$ over the choice of overlays, the DHT allocates any label ℓ to any θ-sized set of nodes with probability at most ε (over the choice of allocation). Note that this definition essentially guarantees a (one-sided) form of load balancing.

Another interesting finding we made was that if the standard scheme is to satisfy our simulation-based definition, then the underlying DHT has to satisfy a form of equivocation. Intuitively, the DHT must be designed in such a way that, for any fixed overlay within a (large) class of overlays, it is possible to "program" the allocation so that it maps a given label to a given server. We found the appearance of equivocation in the context of DHTs quite surprising as it is usually a property that comes up in the context of cryptographic primitives.

Chord in the perpetual setting. Having isolated the properties we need from a DHT in order to prove the security of the standard scheme, it is natural to ask whether there are any known DHTs that satisfy them. Interestingly, we not only found that such DHTs exist but that Chord [78]—which is arguably the most influential DHT—is both balanced and non-committing in the sense that it supports the kind of equivocation discussed above in the random oracle model. Without getting into details of how Chord works (we refer the reader to section 3.5 for a description), we mention here that Chord makes use of two hash functions: one to map names to addresses and a second to map labels to addresses. In section 3.5, we show that Chord is non-committing if the second hash function is modeled as a random oracle.

Transient EDHTs. All the analysis discussed above was for what we call the *perpetual* setting where nodes never leave the network. [1] Note that the perpetual setting is realistic and interesting in itself. It captures, for example, how DHTs are used by many large companies who run nodes in their own data centers, e.g., Amazon, Google, LinkedIn. Nevertheless, we also consider the *transient* setting where nodes are allowed to leave and join the network arbitrarily. We extend our syntax and security definitions to this setting and prove that the standard scheme—equipped with certain join and leave protocols—achieves another probabilistic leakage profile. Necessarily, this leakage profile is more complex than the one achieved in the perpetual setting. At a high level, it works as follows. For puts and gets the leakage is roughly the same as in the perpetual setting. For joins, it leaks the number of previous put operations for labels that were stored and routed exclusively by honest nodes. For leaves there are two cases. When an honest node leaves, the leakage is the same as a join and when a corrupted node leaves there is no leakage. Our leakage analysis in the transient setting relies on a new and stronger property of the underlying DHT we call *stability* which, roughly speaking, means that with probability at least $1 - \delta$ over the choice of overlay parameter ω, for all large enough overlays, the DHT allocates any label to any θ-sized set with probability at most ε.

Chord in the transient setting. Having analyzed the standard EDHT in the transient setting, we study its properties when it is instantiated with a transient variant of Chord. Our analysis of Chord's stability is non-trivial. At a very high level the main challenge is that, in the transient setting, Chord's overlay changes with every leave or join. To handle this, we introduce a series of (probabilistic) bounds to handle "dynamic" overlays that may be of independent interest.

3.2 Related work

Since we already discussed related work on DHTs and their applications, we skip describing it again. However, as described earlier, an EDHT scheme can also be viewed as a form of *distributed* STE scheme, we therefore discuss some related work from the encrypted search literature.

[1] Note that in this setting we allow nodes to fail as long as they come back up in a bounded amount of time.

Encrypted search. An encrypted search algorithm (ESA) is a search algorithm that operates on encrypted data. ESAs can be built from various cryptographic primitives including oblivious RAM (ORAM) [66], fully-homomorphic encryption (FHE) [64], property-preserving encryption (PPE) [25, 33, 38, 100, 85] and structured encryption (STE) [52] which is a generalization of searchable symmetric encryption [107]. Each of these approaches achieves different tradeoffs between efficiency, expressiveness and security/leakage. For large datasets, structured encryption seems to provide the best tradeoffs between these three dimensions: achieving sub-linear (and even optimal) search times and rich queries while leaking considerably less than PPE-based solutions and either the same as [76] or slightly more than ORAM-based solutions. Various aspects of STE have been extensively studied in the cryptographic literature including dynamism [65, 75, 74, 108, 39, 50, 40, 60], locality [49, 28, 56, 29, 57], expressiveness [52, 48, 99, 62, 61, 90, 71, 72, 77, 121] and leakage [69, 47, 76, 37, 73].

3.3 Distributed Hash Tables

A distributed hash table is a distributed storage system that instantiates a dictionary data structure. It is distributed in the sense that the data is stored by a set of n nodes $N_1, \ldots, N_n$ and it instantiates a dictionary in the sense that it stores label/value pairs and supports Get and Put operations. Because they are distributed, DHTs rely on an overlay network which, intuitively, consists of a set of node addresses and a distributed routing protocol. As discussed in Section 5.1, DHTs are a fundamental primitive in distributed systems and have many applications.

In this work, we will consider two kinds of DHTs: perpetual and transient. Perpetual DHTs are composed of a fixed set of nodes that are all known at setup time. They can handle nodes going down (e.g., due to failure) and coming back online but such unresponsive nodes are expected to come back online after some period of time. Transient DHTs, on the other hand, are designed for settings where nodes are not known a-priori and can join and leave at any time. Perpetual DHTs are suitable for "permissioned" settings like the backend infrastructure of large companies whereas transient DHTs are better suited to "permissionless" settings like peer-to-peer networks and permissionless blockchains.

3.3.1 Perpetual DHTs

Syntax. We formalize DHTs as a collection of six algorithms $\mathsf{DHT} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$. The first three algorithms, $\mathsf{Overlay}$, Alloc and $\mathsf{FrontEnd}$ are executed only once by the entity responsible for setting up the system. $\mathsf{Overlay}$ takes as input an integer $n \geq 1$, and outputs a parameter ω from a space Ω. Alloc takes as input parameters ω and n, and outputs a parameter ψ from a space Ψ. $\mathsf{FrontEnd}$ takes as input parameters ω and n and outputs a parameter ϕ from space Φ. We refer to these parameters as the *DHT parameters* and represent them by $\Gamma = (\omega, \psi, \phi)$. Each DHT has an address space $\mathbf{A}$ and the DHT parameters in Γ define different components of the DHT over this address space. For example, ω maps node names to addresses in $\mathbf{A}$, ψ maps labels to addresses in $\mathbf{A}$, ϕ determines the address of a front-end node (or starting node).

The fourth algorithm, Daemon, takes Γ and n as input and is executed by every node in the network. Daemon is halted only when a node wishes to leave the network and it is responsible for setting up its calling node's state for routing messages and for storing and retrieving label/value pairs from the node's local storage. The fifth algorithm, Put, is executed by a client to store a label/value pair on the network. Put takes as input Γ and a label/value pair ℓ and v, and outputs nothing. The sixth algorithm, Get, is executed by a client to retrieve the value associated to a given label from the network. Get takes as input Γ and a label ℓ and outputs a value v. Since all DHT algorithms take Γ as input we sometimes omit it for visual clarity.

Abstracting DHTs. To instantiate a DHT, the parameters ω and ψ must be chosen together with a subset $\mathbf{C} \subseteq \mathbf{N}$ of active nodes (i.e., the nodes currently in the network) and an active set of labels $\mathbf{K} \subseteq \mathbf{L}$ (i.e., the labels stored in the DHT). Once a DHT is instantiated, we can describe it using a tuple of function families $(\mathsf{addr}, \mathsf{server}, \mathsf{route}, \mathsf{fe})$ that are all parameterized by subset of parameters in Γ. These functions are defined as

$$\mathsf{addr}_\omega : \mathbf{N} \to \mathbf{A} \qquad \mathsf{server}_{\omega,\psi} : \mathbf{L} \to \mathbf{A} \qquad \mathsf{route}_\omega : \mathbf{A} \times \mathbf{A} \to 2^{\mathbf{A}}, \qquad \mathsf{fe}_\phi : \mathbf{L} \to \mathbf{A}$$

where addr_ω maps names from a name space $\mathbf{N}$ to addresses from an address space $\mathbf{A}$, $\mathsf{server}_{\omega,\psi}$ maps labels from a label space $\mathbf{L}$ to the address of the node that stores it, route_ω maps two addresses to the addresses of the nodes on the route between them, and fe_ϕ maps labels to node addresses who forward client requests to the rest of the network. For visual clarity we abuse notation and represent the path between two addresses by a *set* of addresses instead of as a sequence of addresses, but we stress that paths are sequences. Note that this is an abstract representation of a DHT that will be particularly useful for our analysis but, in practice, the overlay network, including its addressing and routing functions, are implemented by the Daemon algorithm.

We sometimes refer to a pair $(\omega, \mathbf{C})$ as an overlay and to a pair $(\psi, \mathbf{K})$ as an allocation. Abstractly speaking, we can think of an overlay as an assignment from active nodes to addresses and of an allocation as an assignment of active labels to addresses. In this sense, overlays and allocations are determined by a pair $(\omega, \mathbf{C})$ and $(\psi, \mathbf{K})$, respectively.

Visible addresses. A very useful notion for our purposes will be that of *visible addresses*. For a fixed overlay $(\omega, \mathbf{C})$ an address $a \in \mathbf{A}$ is **s**-visible to a node $N \in \mathbf{C}$ if there exists a label $\ell \in \mathbf{L}$ such that if ψ allocates ℓ to a, then either: (1) $\mathsf{addr}_\omega(N) = \mathsf{server}_{\omega,\psi}(\ell)$; or (2) $\mathsf{addr}_\omega(N) \in \mathsf{route}_\omega(s, \mathsf{server}_{\omega,\psi}(\ell))$. The intuition behind this is that if a label ℓ is mapped to an address in $\mathsf{Vis}(s, N)$ then N either stores the label ℓ or routes it when the operation for ℓ starts at address s. We point out that the visibility of a node changes as we change the starting address s. For example, the node is maybe present on the path to $\mathsf{server}_{\omega,\psi}$ if s is the starting address but not on the path if s' is the starting address. Throughout we assume the set of visible addresses to to be efficiently computable.

Since the set of **s**-visible addresses depends on parameters ω and the set $\mathbf{C}$ of nodes that are currently active, we subscript $\mathsf{Vis}_{\omega,\mathbf{C}}(s, N)$ with all these paramters. Finallym we also extend the

notion to the set of s-visible addresses $\mathsf{Vis}_{\omega,\mathbf{C}}(s, S)$ for a set of nodes $S \subseteq \mathbf{C}$ which is defined simply as $\mathsf{Vis}_{\omega,\mathbf{C}}(s, S) = \cup_{N \in S} \mathsf{Vis}_{\omega,\mathbf{C}}(s, N)$. Again, for visual clarity, we will drop the subscripts wherever they are clear from the context.

Front-end distribution. Another important notion in our analysis is that of a label's *front-end distribution* which is the probability distribution that governs the address of an operation's "entry point" into the DHT network. It is captured by the random variable $\mathsf{fe}_{\phi}(\ell)$, where ϕ is sampled by the algorithm $\mathsf{FrontEnd}$. In this work we assume front-end distributions to be *label-independent* in the sense that every label's front-end node distribution is the same. We therefore simply refer to this distribution as the DHT's front-end distribution.

Allocation distribution. The next important notion in our analysis is what we refer to as a label's *allocation distribution* which is the probability distribution that governs the address at which a label is allocated. More precisely, this is captured by the random variable $\psi(\ell)$, where ψ is sampled by the algorithm Alloc. We also assume allocation distributions are *label-independent* in the sense that every label's allocation distribution is the same [2]. We therefore simply refer to this distribution as the DHT's allocation distribution.

Given a DHT's allocation distribution, we also consider a distribution $\Delta(S)$ that is parameterized by a set of addresses $S \subseteq \mathbf{A}$. This distribution is over S and has probability mass function

$$f_{\Delta(S)}(a) = \frac{f_{\psi}(a)}{\sum_{a \in S} f_{\psi}(a)} = \frac{\Pr\left[\, \psi(\ell) = a \,\right]}{\Pr\left[\, \psi(\ell) \in S \,\right]},$$

where f_{ψ} is the probability mass function of the DHT's allocation distribution.

Non-committing allocations. As we will see in Section 4.5, our EDHT construction can be based on any DHT but the security of the resulting scheme will depend on certain properties of the underlying DHT. We describe these properties here. The first property that we require of a DHT is that the allocations it produces be non-committing in the sense that it supports a form of equivocation. More precisely, for some fixed overlay $(\omega, \mathbf{C})$ and allocation $(\psi, \mathbf{K})$, there should exist some efficient mechanism to arbitrarily change/program ψ. In other words, there should exist a polynomial-time algorithm $\mathsf{Program}$ such that, for all $(\omega, \mathbf{C})$ and $(\psi, \mathbf{K})$, given a label $\ell \in \mathbf{L}$ and address $a \in \mathbf{A}$, $\mathsf{Program}(\ell, a)$ modifies the DHT so that $\psi(\ell) = a$. For the special case of Chord, which we study in Section 3.5, this can be achieved by modeling one of its hash functions as a random oracle.

Balanced overlays. The second property is related to how well the DHT load balances the label/value pairs it stores. While load balancing is clearly important for storage efficiency we will see, perhaps surprisingly, that it also has an impact on security. Intuitively, we say that an overlay $(\omega, \mathbf{C})$ is balanced if for all labels ℓ, the probability that any set of θ nodes sees ℓ is not too large.

[2] This is true for every DHT we are aware of [88, 70, 110, 58].

Definition 3.3.1 (Balanced overlays)**.** *Let $\omega \in \Omega$ be an overlay parameter and let $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes. We say that an overlay $(\omega, \mathbf{C})$ is (ε, θ)-balanced if for all $\ell \in \mathbf{L}$ and for all $S \subseteq \mathbf{C}$ with $|S| = \theta$,*

$$\Pr\left[\,\mathsf{server}_{\omega,\psi}(\ell) \in \mathsf{Vis}_{\omega,\mathbf{C}}(\mathsf{fe}_{\phi}(\ell), S)\,\right] \leq \varepsilon,$$

where the probability is over the coins of Alloc *and* FrontEnd *and where ε can depend on θ.*

Definition 3.3.2 (Balanced DHT)**.** *We say that a distributed hash table* $\mathsf{DHT} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ *is $(\varepsilon, \delta, \theta)$-balanced if for all $\mathbf{C} \subseteq \mathbf{N}$, the probability that an overlay $(\omega, \mathbf{C})$ is (ε, θ)-balanced is at least $1 - \delta$ over the coins of* Overlay *and where ε and δ can depend on $\mathbf{C}$ and θ.*

3.3.2 Transient Distributed Hash Tables

In this section, we formalize DHTs in the context of transient networks.

Syntax. Transient DHTs are a collection of eight algorithms $\mathsf{DHT}^{+} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get}, \mathsf{Leave}, \mathsf{Join})$. The first six algorithms are same as in the perpetual setting. The seventh is an algorithm Leave executed by a node $N \in \mathbf{C}$ when it wishes to leave the network. Leave takes nothing as input and outputs nothing but it halts the Daemon algorithm. The eighth is an algorithm Join that is executed by a node $N \in \mathbf{N} \setminus \mathbf{C}$ that wishes to join the network. It takes nothing as input and outputs nothing but executes the Daemon algorithm. When a node executes a Leave or Join, the routing tables of all the other nodes are updated and label/value pairs are moved around in the network according to allocation ψ. In other words, when a node leaves, its pairs are reallocated in the network and when a node joins, some pairs stored on the other nodes are moved to the new node.

Note that when a node $N \in \mathbf{C}$ leaves the network, the set of active nodes $\mathbf{C}$ automatically shrinks to exclude N. Similarly, when a node $N \in \mathbf{N} \setminus \mathbf{C}$ joins the network, the set of active nodes $\mathbf{C}$ expands to include N. From now on, whenever we write $\mathbf{C}$ we are referring to the current set of active nodes.

Stability. To prove the security of EDHTs in the transient setting, we need the underlying DHT to satisfy a stronger notion than balance which we call *stability*.

Definition 3.3.3 (Stability)**.** *We say that a transient distributed hash table* $\mathsf{DHT}^{+} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEndDaemon}, \mathsf{Put}, \mathsf{Get}, \mathsf{Leave}, \mathsf{Join})$ *is $(\varepsilon, \delta, \theta)$-stable if*

$$\Pr\left[\bigwedge_{\mathbf{C} \subseteq \mathbf{N} : |\mathbf{C}| \geq \theta} (\omega, \mathbf{C}) \text{ is } (\varepsilon, \theta)\text{-balanced}\right] \geq 1 - \delta$$

where the probability is over the choice of ω, and $\varepsilon = \varepsilon(\mathbf{C})$.

Notice that stability requires that Overlay returns an overlay parameter ω such that, with high probability, $(\omega, \mathbf{C})$ is balanced *for all* possible subsets of active nodes $\mathbf{C}$ simultaneously. Balance, on the other hand, only requires that for all sets of active nodes $\mathbf{C}$, with high probability Overlay will output an overlay parameter ω such that $(\omega, \mathbf{C})$ is balanced. In other words, stability requires a single overlay parameter ω that is "good" for all subsets of active nodes whereas balance does not.

3.4 Encrypted Distributed Hash Tables in the Perpetual Setting

In this Section, we formally define encrypted distributed hash tables. An EDHT is an end-to-end encrypted distributed system that instantiates a dictionary data structure.

3.4.1 Syntax and Security Definitions

Syntax. We formalize symmetric EDHTs as a collection of seven algorithms $\mathsf{EDHT} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$. The first algorithm Gen is executed by a client and takes as input a security parameter 1^k and outputs a secret key K. All the other algorithms have the same syntax as before (See Section 3.3), with the difference that Get and Put also take the secret key K as input.

Security. We now turn to formalizing the security of an EDHT. We do this by combining the definitional approaches used in secure multi-party computation [45] and in structured encryption [53, 52]. The security of multi-party protocols is generally formalized using the Real/Ideal-world paradigm. This approach consists of defining two probabilistic experiments **Real** and **Ideal** where the former represents a real-world execution of the protocol where the parties are in the presence of an adversary, and the latter represents an ideal-world execution where the parties interact with a trusted functionality shown in Figure 3.1. The protocol is secure if no environment can distinguish between the outputs of these two experiments. Below, we will describe both these experiments more formally.

Before doing so, we discuss an extension to the standard definitions. To capture the fact that a protocol could leak information to the adversary, we parameterize the definition with a leakage profile that consists of a leakage function $\mathcal{L}$ that captures the information leaked by the Put and Get operations. Our motivation for making the leakage explicit is to highlight its presence.

The real-world experiment. The experiment is executed between a trusted party $\mathcal{T}$, a client $\mathcal{C}$, a set $\mathbf{C} \subseteq \mathbf{N}$ of n nodes $N_1, \ldots, N_n$, an environment $\mathcal{Z}$ and an adversary $\mathcal{A}$. The trusted party $\mathcal{T}$ runs $\mathsf{Overlay}(n)$ and $\mathsf{Alloc}(\omega, n)$ and $\mathsf{FrontEnd}(\omega, n)$, and sends (ω, ψ, ϕ) to all parties, i.e., the nodes, the client, the environment and the adversary. Given $z \in \{0,1\}^*$, the environment $\mathcal{Z}$ sends to the adversary $\mathcal{A}$, a subset $I \subseteq \mathbf{C}$ of nodes to corrupt. The client $\mathcal{C}$ generates a secret key $K \leftarrow \mathsf{Gen}(1^k)$. The nodes execute $\mathsf{EDHT.Daemon}(\omega, \psi, n)$. $\mathcal{Z}$ then adaptively chooses a polynomial number of

Functionality $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}$

$\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}$ stores a dictionary DX initialized to empty and proceeds as follows, running with client $\mathcal{C}$, n nodes $N_1, \ldots, N_n$ and a simulator Sim:

- $\mathsf{Put}(\ell, v)$: Upon receiving a label/value pair (ℓ, v) from client $\mathcal{C}$, it sets $\mathsf{DX}[\ell] := v$, and sends the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{put}, \ell, v))$ to the simulator Sim.
- $\mathsf{Get}(\ell)$: Upon receiving a label ℓ from client C, it returns $\mathsf{DX}[\ell]$ to the client $\mathcal{C}$ and the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{get}, \ell, \perp))$ to the simulator Sim.

Figure 3.1: $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}$: The DHT functionality parameterized with leakage function $\mathcal{L}$.

operations op_j, where $\mathsf{op}_j \in \{\mathtt{get}, \mathtt{put}\} \times \mathbf{L} \times \{\mathbf{V}, \perp\}$ and sends it to $\mathcal{C}$. If $\mathsf{op}_j = (\mathtt{get}, \ell)$, the client $\mathcal{C}$ executes $\mathsf{EDHT.Get}(K, \ell)$. If $\mathsf{op}_j = (\mathtt{put}, \ell, v)$, $\mathcal{C}$ initiates $\mathsf{EDHT.Put}(K, \ell, v)$. The client forwards its output from running the get/put operations to $\mathcal{Z}$. $\mathcal{A}$ computes a message m from its view and sends it to $\mathcal{Z}$. Finally, $\mathcal{Z}$ returns a bit that is output by the experiment. We let $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ be a random variable denoting $\mathcal{Z}$'s output bit.

The ideal-world experiment. The experiment is executed between a client $\mathcal{C}$, a set $\mathbf{C} \subseteq \mathbf{N}$ of n nodes $N_1, \ldots, N_n$, an environment $\mathcal{Z}$ and a simulator Sim. Each party also has access to the ideal functionality $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}$. Given $z \in \{0,1\}^*$, the environment $\mathcal{Z}$ sends to the simulator Sim, a subset $I \subseteq \mathbf{C}$ of nodes to corrupt. $\mathcal{Z}$ then adaptively chooses a polynomial number of operations op_j, where $\mathsf{op}_j \in \{\mathtt{get}, \mathtt{put}\} \times \mathbf{L} \times \{\mathbf{V}, \perp\}$, and sends it to the client $\mathcal{C}$ which, in turn, forwards it to $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}$. If $\mathsf{op}_j = (\mathtt{get}, \ell)$, the functionality executes $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}.\mathsf{Get}(\ell)$. Otherwise, if $\mathsf{op}_j = (\mathtt{put}, \ell, v)$ the functionality executes $\mathcal{F}_{\mathsf{DHT}}^{\mathcal{L}}.\mathsf{Put}(\ell, v)$. $\mathcal{C}$ forwards its outputs to $\mathcal{Z}$ whereas Sim sends $\mathcal{Z}$ some arbitrary message m. Finally, $\mathcal{Z}$ returns a bit that is output by the experiment. We let $\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k)$ be a random variable denoting $\mathcal{Z}$'s output bit.

Definition 3.4.1 ($\mathcal{L}$-security)**.** *We say that an encrypted distributed hash table* $\mathsf{EDHT} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ *is $\mathcal{L}$-secure, if for all* PPT *adversaries $\mathcal{A}$ and all* PPT *environments $\mathcal{Z}$, there exists a* PPT *simulator* Sim *such that for all $z \in \{0,1\}^*$,*

$$|\Pr[\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k) = 1]| \leq \mathsf{negl}(k).$$

3.4.2 The Standard EDHT in the Perpetual Setting

We now describe the standard approach to storing sensitive data on a DHT. This approach relies on simple cryptographic primitives and a non-committing and balanced DHT.

Overview. The scheme $\mathsf{EDHT} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ is described in detail in Figure 3.2 and, at a high level, works as follows. It makes black-box use of a distributed hash table $\mathsf{DHT} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$, a pseudo-random function F and a symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

Let $\mathsf{DHT} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ be a distributed hash table, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme and F be a pseudo-random function. Consider the encrypted distributed hash table $\mathsf{EDHT} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ that works as follows:

- $\mathsf{Gen}(1^k)$:
  1. sample $K_1 \xleftarrow{\$} \{0,1\}^k$ and compute $K_2 \leftarrow \mathsf{SKE.Gen}(1^k)$
  2. output $K = (K_1, K_2)$
- $\mathsf{Overlay}(n)$:
  1. compute and output $\omega \leftarrow \mathsf{DHT.Overlay}(n)$
- $\mathsf{Alloc}(n, \omega)$:
  1. compute and output $\psi \leftarrow \mathsf{DHT.Alloc}(n, \omega)$
- $\mathsf{FrontEnd}(n, \omega)$:
  1. compute and output $\phi \leftarrow \mathsf{DHT.FrontEnd}(n, \omega)$
- $\mathsf{Daemon}(\omega, \psi, n)$:
  1. Execute $\mathsf{DHT.Daemon}(\omega, \psi, n)$
- $\mathsf{Put}(K, \ell, v)$:
  1. Parse K as (K_1, K_2)
  2. compute $t := F_{K_1}(\ell)$
  3. compute $e \leftarrow \mathsf{SKE.Enc}(K_2, v)$
  4. execute $\mathsf{DHT.Put}(t, e)$
- $\mathsf{Get}(K, \ell)$:
  1. Parse K as (K_1, K_2)
  2. Initialise $v := \perp$
  3. compute $t := F_{K_1}(\ell)$
  4. execute $e \leftarrow \mathsf{DHT.Get}(t)$
  5. if $e \neq \perp$, compute and output $v \leftarrow \mathsf{SKE.Dec}(K_2, e)$

Figure 3.2: EDHT: The Standard EDHT Scheme

The Gen algorithm takes as input a security parameter 1^k and uses it to generate a key K_1 for the pseudo-random function F and a key K_2 for the symmetric encryption scheme SKE. It then outputs a key $K = (K_1, K_2)$. The Overlay, Alloc, FrontEnd and Daemon algorithms respectively execute DHT.Overlay, DHT.Alloc, DHT.FrontEnd and DHT.Daemon to generate and output the paramters ω, ψ and ϕ. The Put algorithm takes as input the secret key K and a label/value pair (ℓ, v). It first computes $t := F_{K_1}(\ell)$ and $e \leftarrow \mathsf{Enc}(K_2, v)$ and then executes $\mathsf{DHT.Put}(t, e)$. The Get algorithm takes as input the secret key K and a label ℓ. It computes $t := F_{K_1}(\ell)$ and executes $e \leftarrow \mathsf{DHT.Get}(t)$. It then outputs $\mathsf{SKE.Dec}(K, e)$.

Security. We now describe the leakage of EDHT. Intuitively, it reveals to the adversary the times at which a label is stored or retrieved with some probability. More formally, it is defined with the

following *stateful* leakage function

- $\mathcal{L}_\varepsilon(\mathsf{DX}, (\mathsf{op}, \ell, v))$:
  1. if ℓ has never been seen
     (a) sample and store $b_\ell \leftarrow \mathsf{Ber}(\varepsilon)$
  2. if $b_\ell = 1$
     (a) if $\mathsf{op} = \mathtt{put}$ output $(\mathtt{put}, \mathsf{opeq}(\ell))$
     (b) else if $\mathsf{op} = \mathtt{get}$ output $(\mathtt{get}, \mathsf{opeq}(\ell))$
  3. else if $b_\ell = 0$
     (a) output $\perp$

where opeq is the *operation equality pattern* which reveals if and when a label was queried or put in the past. Note that when $\varepsilon = 1$ (for some θ), $\mathcal{L}_\varepsilon$ reduces to the leakage profile achieved by standard encrypted dictionary constructions [52, 50]. On the other hand, when $\varepsilon < 1$, this leakage profile is "better" than the profile of known constructions.

Discussion. We now explain why the leakage function is probabilistic and why it depends on the balance of the underlying DHT. Intuitively, one expects that the adversary's view is only affected by get and put operations on labels that are either: (1) allocated to a corrupted node; or (2) allocated to an uncorrupted node whose path (starting from the client) includes a corrupted node. In such a case, the adversary's view would not be affected by all operations but only a subset of them. Our leakage function captures this intuition precisely and it is probabilistic because, in the real world, the subset of operations that affect the adversary's view is determined probabilistically because it depends on the choice of overlay and allocation—both of which are chosen at random. The way this is handled in the leakage function is by sampling a bit b with some probability and revealing leakage on the current operation if $b = 1$. This determines the subset of operations whose leakage will be visible to the adversary.

Now, for the simulation to go through, the operations simulated by the simulator need to be visible to the adversary with the same probability as in the real execution. But these probabilities depend on ω and ψ which are not known to the leakage function. Note that this implies a rather strong definition in the sense that the scheme hides information about the overlay and the allocation of the DHT.

Since ω and ψ are unknown to the leakage function, the leakage function can only guess as to what they could be. But because the DHT is guaranteed to be $(\varepsilon, \delta, \theta)$-balanced, the leakage function can assume that, with probability at least $1 - \delta$, the overlay will be (ε, θ)-balanced which, in turn, guarantees that the probability that a label is visible to any adversary with at most θ corruptions is at most ε. Therefore, in our leakage function, we can set the probability that $b = 1$ to be ε in the hope that simulator can "adjust" the probability internally to be in accordance to the ω that it sampled. Note that the simulator can adjust the probability only if for its own chosen

ω, the probability that a query is visible to the adversary is less than ε. But this will happen with probability at least $1-\delta$ so the simulation will work with probability at least $1-\delta$.

We are now ready to state our main security Theorem which proves that the standard EDHT construction is $\mathcal{L}_\varepsilon$-secure with probability that is negligibly close to $1-\delta$ when its underlying DHT is $(\varepsilon, \delta, \theta)$-balanced.

Theorem 3.4.2. *If $|I| \leq \theta$ and if* DHT *is $(\varepsilon, \delta, \theta)$-balanced, has non-committing and has label-independent allocation and front-end distributions, then* EDHT *is $\mathcal{L}_\varepsilon$-secure with probability at least $1 - \delta - \mathsf{negl}(k)$.*

Proof. Consider the simulator Sim that works as follows. Given a set of corrupted nodes $I \subseteq \mathbf{C}$, it computes $\omega \leftarrow \mathsf{DHT.Overlay}(n)$, initializes n nodes $N_1, \ldots, N_n$ in $\mathbf{C}$, simulates the adversary $\mathcal{A}$ with I as input and generates a symmetric key $K \leftarrow \mathsf{SKE.Gen}(1^k)$. In the following, let $B \stackrel{def}{=} \mathsf{Vis}_{\omega,\mathbf{C}}(s, I)$ and $p' \stackrel{def}{=} \Pr[\,\psi(\ell) \in B\,]$, which is unique since we assume label-independent allocations. If $p' > \varepsilon$, the simulator aborts otherwise it continues.

When a put/get operation is executed, Sim receives from $\mathcal{F}_{\mathsf{EDHT}}$ the leakage

$$\lambda \in \left\{ \Big(\mathtt{put}, \mathsf{opeq}(\ell)\Big), \Big(\mathtt{get}, \mathsf{opeq}(\ell)\Big), \perp \right\}.$$

If $\lambda = \perp$ then Sim does nothing. If $\lambda \neq \perp$, then Sim checks the query equality to see if the label has been used in the past. If not, it samples and stores a bit

$$b' \leftarrow \mathsf{Ber}\left(\frac{p'}{\varepsilon}\right).$$

Note that, this is indeed a valid Bernoulli distribution since

$$p' = \Pr[\,\psi(\ell) \in B\,] = \Pr[\,\mathsf{server}_{\omega,\psi}(\ell) \in \mathsf{Vis}_{\omega,\mathbf{C}}(s, I)\,] \leq \varepsilon,$$

where the second equality follows from the definition of visible address, and the last inequality follows from $|I| \leq \theta$ and $(\omega, \mathbf{C})$ being (ε, θ)-balanced.

If the label was seen in the past, Sim retrieves the bit b' that was previously sampled. If $b' = 0$, then it does nothing, but if $b' = 1$ it uses the query equality to check if the label has been used in the past. If so, it sets t to the d-bit value previously used. If not, it sets $t \stackrel{\$}{\leftarrow} \{0,1\}^d$, computes $e \leftarrow \mathsf{SKE.Enc}(K, 0)$, and samples an address $a \leftarrow \Delta_{\omega,\mathbf{C}}(B)$, and programs ψ to map t to a. Finally, if the operation was a put, it executes $\mathsf{DHT.Put}(t, e)$, otherwise it executes $\mathsf{DHT.Get}(t)$. Once all of the environment's operations are processed, the simulator returns whatever the adversary outputs.

It remains to show that the view of the adversary $\mathcal{A}$ during the simulation is indistinguishable from its view in a **Real** experiment. We do this using a sequence of games.

Game_0 : is the same as a $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ experiment.

Game_1 : is the same as Game_0 except that the encryption of the value v during a Put is replaced by $\mathsf{SKE.Enc}(K_2, 0)$.

Game_2 : is the same as Game_1 except that output of the PRF F is replaced by a truly random string of d bits.

Game_3 : is the same as Game_2 except that for each operation (op, ℓ, v) (where v can be null), we check if ℓ has been seen before. If not, we sample a bit $b_\ell \leftarrow \mathsf{Ber}(\varepsilon)$, else we set b_ℓ to the bit previously sampled. If $b_\ell = 1$ and $\mathsf{op} = (\mathtt{put}, \ell, v)$, we replace the Put operation with $\mathsf{Sim}(\mathtt{put}, \mathsf{opeq}(\ell))$, and if $b_\ell = 1$ and $\mathsf{op} = (\mathtt{get}, \ell)$, we replace the Get operation with $\mathsf{Sim}(\mathtt{get}, \mathsf{opeq}(\ell))$. If $b_\ell = 0$, we do nothing.

Game_1 is indistinguishable from Game_0, otherwise the encryption scheme is not semantically secure. Game_2 is indistinguishable from Game_1 because the outputs of pseudorandom functions are indistinguishable from random strings.

We now show that the adversary's views in Game_2 and Game_3 are indistinguishable. We denote these views by $\mathbf{view_2}(I)$ and $\mathbf{view_3}(I)$, respectively, and consider the ith "sub-views" $\mathbf{view_2}^i(I)$ and $\mathbf{view_3}^i(I)$ which include the set of messages seen by the adversary (through the corrupted nodes) during the execution of op_i. Let $\mathbf{op}$ denote the sequence of q operations generated by the environment. Let $\ell_1, \ldots, \ell_q$ be the labels of the operations in $\mathbf{op}$, and let $t_1, \ldots, t_q$ be the corresponding random strings obtained by replacing $F_K(\ell_i)$ with random strings. Because DHT is $(\varepsilon, \delta, \theta)$-balanced, we know that with probability at least $1 - \delta$, the overlay $(\omega, \mathbf{C})$ will be (ε, θ)-balanced. So for the remainder of the proof, we assume the overlay is (ε, θ)-balanced.

First, we treat the case where t_i (or equivalently ℓ_i) has never been seen before. Let $\mathcal{E}_i$ be the event that $\psi(t_i) \in B$, where $B = \mathsf{Vis}_{\omega,\mathbf{C}}(s, I)$ are the addresses visible to the corrupted nodes. For all possible views $\mathbf{v}$, we have

$$\begin{aligned}
&\Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v}\,\right] \\
&\quad= \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \wedge \mathcal{E}_i\,\right] + \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \wedge \overline{\mathcal{E}_i}\,\right] \\
&\quad= \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \mathcal{E}_i\,\right] \cdot \Pr\left[\,\mathcal{E}_i\,\right] + \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \overline{\mathcal{E}_i}\,\right] \cdot \Big(1 - \Pr\left[\,\mathcal{E}_i\,\right]\Big) \\
&\quad= \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \mathcal{E}_i\,\right] \cdot \Pr\left[\,\mathcal{E}_i\,\right]
\end{aligned}$$

where the third equality follows from the fact that, conditioned on $\overline{\mathcal{E}_i}$, the nodes in I do not see any messages at all.

Turning to $\mathbf{view_3}$, let $\mathcal{Q}_i$ be the event that $b_i = 1 \wedge b'_i = 1$. Then for all possible views $\mathbf{v}$, we

have

$$\begin{aligned}
&\Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v}\,\right] \\
&\quad = \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \wedge \mathcal{Q}_i\,\right] + \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \wedge \overline{\mathcal{Q}_i}\,\right] \\
&\quad = \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \mid \mathcal{Q}_i\,\right] \cdot \Pr\left[\,\mathcal{Q}_i\,\right] + \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \overline{\mathcal{Q}_i}\,\right] \cdot \Big(1 - \Pr\left[\,\mathcal{Q}_i\,\right]\Big) \\
&\quad = \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \mid \mathcal{Q}_i\,\right] \cdot \Pr\left[\,\mathcal{Q}_i\,\right] \qquad (3.1)
\end{aligned}$$

where the third equality follows from the fact that, for all i, conditioned on $\overline{\mathcal{Q}_i}$, either Sim is never executed or Sim does nothing. In either case, the nodes in I will not see any messages, so for all $\mathbf{v}$ we have $\Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \mid \overline{\mathcal{Q}_i}\,\right] = 0$.

Notice, however, that

$$\Pr\left[\,\mathcal{Q}_i\,\right] = \Pr\left[\,b_i = 1 \wedge b_i' = 1\,\right] = \varepsilon \cdot \frac{\Pr\left[\,\psi(t_i) \in B\,\right]}{\varepsilon} = \Pr\left[\,\psi(t_i) \in B\,\right] = \Pr\left[\,\mathcal{E}_i\,\right],$$

so to show that the views are equally distributed it remains to show that for all $\mathbf{v}$,

$$\Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \mathcal{E}_i\,\right] = \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \mid \mathcal{Q}_i\,\right]. \qquad (3.2)$$

To see why this holds, notice that, conditioned on $\mathcal{E}_i$ and $\mathcal{Q}_i$, the only difference between Game_2 and Game_3 is that, in the former, the labels t_i are mapped to an address a according to an allcoation $(\psi, \mathbf{K})$ generated using Alloc, whereas in the latter, the labels t_i are programmed to an address a sampled from $\Delta_{\omega,\mathbf{C}}(B)$. We show, however, that in both cases, the labels t_i are allocated with the same probability distribution. In Game_2, for all $a \in B$, we have

$$\Pr\left[\,\psi(t_i) = a \mid \mathcal{E}_i\,\right] = \frac{\Pr\left[\,\psi(t_i) = a \wedge \mathcal{E}_i\,\right]}{\Pr\left[\,\mathcal{E}_i\,\right]} = \frac{\Pr\left[\,\psi(t_i) = a\,\right]}{\Pr\left[\,\mathcal{E}_i\,\right]} = \frac{\Pr\left[\,\psi(t_i) = a\,\right]}{\Pr\left[\,\psi(t_i) \in B\,\right]},$$

where the second equality follows from the fact that the event $\big\{\psi(t_i) = a\big\} \subseteq \mathcal{E}_i$.
In Game_3, for all $a \in B$, we have,

$$\Pr\left[\,\psi(t_i) = a \mid \mathcal{Q}_i\,\right] = \frac{\Pr\left[\,\psi(t_i) = a\,\right]}{\Pr\left[\,\psi(t_i) \in B\,\right]},$$

since a is sampled from $\Delta_{\omega,\mathbf{C}}(B)$. Since, for all i, conditioned on $\mathcal{Q}_i$ and E_i, labels are allocated to addresses with the same distribution in both games and since this is the only difference between the games,

$$\Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v} \mid \mathcal{Q}_i\,\right] = \Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v} \mid \mathcal{E}_i\,\right]. \qquad (3.3)$$

Plugging Eq. 3.3 into Eq. 3.1, we have that for all i and all $\mathbf{v}$,

$$\Pr\left[\,\mathbf{view_2}^i(I) = \mathbf{v}\,\right] = \Pr\left[\,\mathbf{view_3}^i(I) = \mathbf{v}\,\right].$$

Now we consider the case where t_i has been seen in the past. In this case, Put or Get operations will produce the same messages that were generated in the past which means that $\mathbf{view_2}^i(I)$ will be the same as before. Similarly, $\mathbf{view_3}^i(I)$ will be the same as before because, whenever t_i has been seen in the past, Sim behaves the same.

■

Efficiency. The standard scheme does not add any overhead to time, round, communication and storage complexities of the underlying DHT.

3.5 A Chord-Based EDHT in the Perpetual Setting

In this section, we analyze the security of the standard EDHT when its underlying DHT is instantiated with Chord. We first give a brief overview of how Chord works and then show that: (1) it has non-committing overlays in the random oracle model; and (2) it is balanced.

Setting up Chord. For Chord, the space Ω is the set of all hash functions $\mathcal{H}_1$ from $\mathbf{N}$ to $\mathbf{A} = \{0, \ldots, 2^m - 1\}$. $\mathsf{Overlay}$ samples a hash function H_1 uniformly at random from $\mathcal{H}_1$ and outputs $\omega = H_1$. The map addr_ω is the hash function itself so Chord assigns to each active node $N \in \mathbf{C}$ an address $H_1(N)$ in $\mathbf{A}$. We call the set $\chi_{\mathbf{C}} = \{H_1(N_1), \ldots, H_1(N_n)\}$ of addresses assigned to active nodes a *configuration*.

The parameter space Ψ is the set of all hash functions $\mathcal{H}_2$ from $\mathbf{L}$ to $\mathbf{A} = \{0, \ldots, 2^m - 1\}$. Alloc samples a hash function H_2 uniformly at random from $\mathcal{H}_2$ and outputs $\psi = H_2$. The map $\mathsf{server}_{\omega,\psi}$ maps every label ℓ in $\mathbf{L}$ to the address of the active node that follow $H_2(\ell)$ in a clockwise direction. More formally, $\mathsf{server}_{\omega,\psi}$ is the function $\mathsf{succ}_{\chi_{\mathbf{C}}} \circ H_2$, where $\mathsf{succ}_{\chi_{\mathbf{C}}}$ is the *successor* function that assigns each address in $\mathbf{A}$ to its least upper bound in $\chi_{\mathbf{C}}$. Here, $\{0, \ldots, 2^m - 1\}$ is viewed as a "ring" in the sense that the successor of 2^{m-1} is 0.

Chord allows its clients to choose any node as the front-end node to issue its operations. Moreover, it does not restrict them to connect to the same node $\mathsf{fe}_\phi(\ell)$, everytime the client wants to query the same ℓ. This means that for Chord, fe_ϕ is not necessarily a function but can be a one-to-many relation. Unfortunately we cannot prove Chord to be balanced for arbitrary fe_ϕs. We therefore modify Chord and model its space Φ as the set of all hash functions $\mathcal{H}_3$ from $\mathbf{L}$ to addresses of active nodes. $\mathsf{FrontEnd}$ samples a hash function H_3 uniformly at random from $\mathcal{H}_3$ and outputs $\phi = H_3$. The map fe_ϕ is the hash function H_3 itself so it assigns a front-end node with address $H_3(\ell)$ to each label ℓ.

Based on $\omega = H_1$, the Daemon algorithm constructs a routing table by storing the addresses of the node's 2^ith successor where $0 \leq i \leq \log n$ (we refer the reader to [110] for more details). Note that a routing table contains at most $\log n$ other nodes. The Chord routing protocol is fairly simple: given a message destined to a node N_d, a node N checks if $N = N_d$. If not, the node forwards the message to the node N' in its routing table with an address closest to N_d. Note that the route_ω map for Chord is deterministic given a fixed set of active nodes and it guarantees that any two nodes have a path of length at most $\log n$.

Storing and retrieving. Once the DHT is instantiated, each Chord node instantiates an empty dictionary data structure DX_i. When a client executes a Put operation on a label/value pair (ℓ, v),

it computes $N_\ell = \mathsf{succ}_{\chi_\mathbf{C}}(H_2(\ell))$ and uses the Chord routing protocol to send the pair (ℓ, v) to the node N_ℓ who stores it in its local dictionary DX_i. When executing a Get query on a label ℓ, the Client also computes $N_\ell = \mathsf{succ}_{\chi_\mathbf{C}}(H_2(\ell))$ and, again, uses the Chord routing protocol to send the label ℓ to N_ℓ. The latter looks up ℓ in its local dictionary DX_i and uses the Chord routing protocol to return the associated value v.

Non-committing allocation. Given a label ℓ and an address θ, the allocation $(H_2, \mathbf{K})$ can be changed by programming the random oracle H_2 to output θ when it is queried on ℓ.

Allocation distribution. Since Chord assigns labels to addresses using a random oracle H_2, it follows that for all overlays $(H_1, \mathbf{C})$, all labels $\ell \in \mathbf{L}$ and all addresses $a \in \mathbf{A}$,

$$f_{H_2}(a) = \Pr\left[\, H_2(\ell) = a \,\right] = \frac{1}{|\mathbf{A}|},$$

which implies that Chord has label-independent allocations. From this it also follows that $\Delta(S)$ has a probability mass function

$$f_{\Delta(S)}(a) = \frac{1}{|S|}.$$

Before describing the visibility of nodes in Chord and analyzing its balance, we define notation that will be useful in our analysis.

Notation. The *arc* of a node N is the set of addresses in $\mathbf{A}$ between N's predecessor and itself. Note that the arc of a node depends on a configuration χ. More formally, we write $\mathsf{arc}_\chi(N) = (\mathsf{pred}_\chi(H_1(N)), \ldots, H_1(N)]$, where $\mathsf{pred}_\chi(N)$ is the *predecessor* function which assigns each address in $\mathbf{A}$ to its largest lower bound in χ. The *area* of a node N is defined as $\mathsf{area}(\chi, N) = |\mathsf{arc}_\chi(N)|$ and the area of a set of nodes $S \subseteq \chi$ is $\mathsf{area}(\chi, S) = \sum_{N \in S} \mathsf{area}(\chi, N)$. We denote by $\mathsf{maxareas}(\chi, x)$, the sum of the areas of x largest arcs in configuration χ. The *maximum area* of a configuration χ is equal to $\mathsf{maxareas}(\chi, \theta)$. As we will see, the maximum area is central not only to analyzing the balance of Chord but also to analyzing its stability.

Visible addresses. Given a fixed overlay $(H_1, \mathbf{C})$, an address $s \in \mathbf{A}$ and a node $N \in \mathbf{C}$, if the starting address is $s = H_1(N)$, then $\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) = \mathbf{A}$. This is because $H_1(N)$ lies on $\mathsf{route}_{\chi_\mathbf{C}}(s, a)$ for all $a \in \mathbf{A}$. Now for an address $s \in \mathbf{A}$ such that $s \neq H_1(N)$, we have

$$\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) = \left\{ \mathsf{arc}_{\chi_\mathbf{C}}(N') : H_1(N) \in \mathsf{route}_{\chi_\mathbf{C}}(s, H_1(N')) \right\} \bigcup \mathsf{arc}_{\chi_\mathbf{C}}(N)$$

Finally, for any set $S \subseteq \mathbf{C}$, $\mathsf{Vis}_{\omega,\mathbf{C}}(s, S) = \cup_{N \in S} \mathsf{Vis}_{\omega,\mathbf{C}}(s, N)$.

3.5.1 Analyzing Chord's Maximum Area

As we showed in Theorem 3.4.2, the leakage profile of the standard EDHT depends on the balance of the underlying DHT. As we will see, analyzing the balance of Chord is non-trivial and relies on

a quantity we call the *maximum area*. Before defining and analyzing this quantity we first describe some notation.

Preliminaries. We now recall a Theorem from Byers, Considine and Mitzenmacher [42] that will help us upper bound Chord's maximum area.

Theorem 3.5.1 ([42])**.** *Let $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes. If the following conditions hold (where all the probabilities are over the coins of* Overlay*):*

1. *for some constant δ_1,*

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{C}}, 1) \leq \frac{\delta_1 |\mathbf{A}| \log |\mathbf{C}|}{|\mathbf{C}|}\right] \leq 1 - p_1$$

2. *For suitable constants $\delta_2, \delta_3, \delta_4 > 0$, and $2 \leq c \leq \delta_4 \log |\mathbf{C}|$,*

$$\Pr\left[\left|\left\{\alpha \in A : |\alpha| \geq \frac{c|\mathbf{A}|}{|\mathbf{C}|}\right\}\right| \leq \frac{\delta_2 |\mathbf{C}|}{e^{c/\delta_3}}\right] \geq 1 - p_2$$

   where A is the set of all arcs in $\chi_{\mathbf{C}}$.

then, for all $\theta \leq c_2 |\mathbf{C}|$

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \frac{\gamma_1 |\mathbf{A}| \theta}{|\mathbf{C}|} \log \frac{|\mathbf{C}|}{\theta}\right] \geq 1 - p_1 - p_2 \cdot \log |\mathbf{C}|$$

where

$$\gamma_1 = 2\delta_3 + \frac{\delta_1}{1 - \frac{\delta_4}{2\delta_3}}, \quad \text{and} \quad c_2 = \min(2\delta_2 e^{-2/\delta_3}, 1/e).$$

To use Theorem 3.5.1 to bound Chord's maximum area, we need to find the constants for which Chord satisfies the Theorem's two conditions. We do this using the following Lemmas. The first is by Wang and Loguinov [117] and upper bounds the size of Chord's maximum arc (i.e. $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$).

Lemma 3.5.2 ([117])**.** *Let $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes. Then,*

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{C}}, 1) \leq \frac{(1 + c_1)|\mathbf{A}| \log |\mathbf{C}|}{|\mathbf{C}|}\right] \geq 1 - \frac{1}{|\mathbf{C}|^{c_1}},$$

where the probability is over the coins of Overlay *(i.e., the choice of H_1).*

For the second condition, we recall another Lemma from [42] based on the negative dependence of the size of Chord's arcs.

Lemma 3.5.3 ([42])**.** *Let $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes. For $2 \leq c \leq n$,*

$$\Pr\left[\left|\left\{\alpha \in A : |\alpha| \geq \frac{c|\mathbf{A}|}{|\mathbf{C}|}\right\}\right| \geq \frac{2|\mathbf{C}|}{e^c}\right] \leq e^{-|\mathbf{C}|e^{-c/3}}$$

where the probability is over the coins of Overlay *(i.e., the choice of H_1).*

Finding the constants. From Theorem 3.5.1 and Lemmas 3.5.2 and 3.5.3, we have the following Corollary which upper bounds Chord's maximum area.

Corollary 3.5.4. *Let* $\mathbf{C} \subseteq \mathbf{N}$ *be a set of active nodes. Then, for* $\theta \leq |\mathbf{C}|/e$

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \frac{6|\mathbf{A}|\theta}{|\mathbf{C}|}\log\frac{|\mathbf{C}|}{\theta}\right] \geq 1 - \frac{1}{|\mathbf{C}|^2} - (e^{-\sqrt{|\mathbf{C}|}} \cdot \log|\mathbf{C}|).$$

Proof. Setting $c_1 = 2$ in Lemma 3.5.2, we get $\delta_1 = 3$, and $p_1 = 1/|\mathbf{C}|^2$. Setting $c = \delta_4 \log|\mathbf{C}|$ in Lemma 3.5.3, we get $\delta_2 = 2$, $\delta_3 = 1$, and $p_2 \approx e^{-|\mathbf{C}|^{1-\delta_4}} = e^{-\sqrt{|\mathbf{C}|}}$ (setting $\delta_4 = 0.5$). Substituting the values of $\delta_1 = 3$, $\delta_2 = 2$, $\delta_3 = 1$, $\delta_4 = 0.5$, we get $\gamma_1 = 6$ and $\gamma_2 = e$.
Therefore, from Theorem 3.5.1, for all $\theta \leq |\mathbf{C}|/e$,

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \frac{6|\mathbf{A}|\theta}{|\mathbf{C}|}\log\frac{|\mathbf{C}|}{\theta}\right] \geq 1 - \frac{1}{|\mathbf{C}|^2} - (e^{-\sqrt{|\mathbf{C}|}} \cdot \log|\mathbf{C}|)$$

■

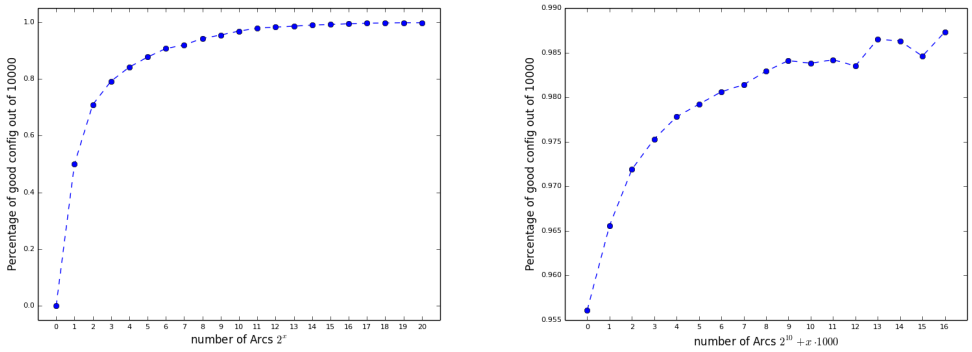
Experimental evaluation of maximum area. In the above Corollary, the error probability of $O(1/|\mathbf{C}|^2)$ stems from the fact that Lemma 3.5.2 only bounds $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$ with probability $1 - O(1/|\mathbf{C}|^2)$.

We ran two experiments to empirically study the probability that $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$ is bounded by $(1 + c_1)|\mathbf{A}|\log|\mathbf{C}|/|\mathbf{C}|$. We found out that probability of sampling a good configuration is approximately 0.96 when $|\mathbf{C}| \approx 2^{10} = 1024$. Therefore, our experiments suggest that for $|\mathbf{C}| \geq 1024$, the Overlay algorithm samples a good configuration with exponentially high probability (See Figure 3.3). In both experiments, we set $|\mathbf{A}| = 2^{24}$ and $c_1 = 0$ and vary $|\mathbf{C}|$. Then, for each value of $|\mathbf{C}|$, we sample 10000 configurations as follows: we sample $|\mathbf{C}|$ points uniformly at random from $\mathbf{A}$, sort them and compute the length of the maximum arc $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$. We then count the number of configurations for which $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1) \leq (1 + c_1)|\mathbf{A}|\log|\mathbf{C}|/|\mathbf{C}| = |\mathbf{A}|\log|\mathbf{C}|/|\mathbf{C}|$. We call such configurations "good" configurations. This gives us the probability of sampling a good configuration for fixed $|\mathbf{A}|$, c_1 and $|\mathbf{C}|$. Note that we chose $c_1 = 0$ because this is the worst value of c_1: any configuration with $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$ less than $|\mathbf{A}|\log|\mathbf{C}|/|\mathbf{C}|$ will also have $\mathsf{maxareas}(\chi_{\mathbf{C}}, 1)$ less than $(1 + c_1)|\mathbf{A}|\log|\mathbf{C}|/|\mathbf{C}|$, where $c_1 \geq 1$.

Figure 3.3a shows the probability of sampling a good configuration as the number of nodes (or correspondingly arcs) are doubled from 1 to 2^{20}, and Figure 3.3b shows the probability when the number of nodes are incremented by 1000 starting from 2^{10} until $\sim 2^{14}$. We see in both plots of Figure 3.3 that the probability of sampling a good configuration increases exponentially as a function of the number of active nodes $|\mathbf{C}|$. Moreover, the probability of sampling a good configuration is approximately 0.96 when $|\mathbf{C}| \approx 2^{10} = 1024$. Therefore, our experiments suggest that for $|\mathbf{C}| \geq 1024$, the Overlay algorithm samples a good configuration with exponentially high probability.

3.5.2 The Balance of Chord

We are now ready to analyze the balance of Chord.



(a) (b)

Figure 3.3: Probability of sampling a good configuration. We write a value x on x-axis to mean 2^x in Figure 3.3a while $2^{10} + x \cdot 1000$ in Figure 3.3b.

Theorem 3.5.5. *Let* $\mathbf{C} \subseteq \mathbf{N}$ *be a set of active nodes. If* $\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \lambda$*, then* $\chi_{\mathbf{C}}$ *is* (ε, θ)*-balanced with*

$$\varepsilon = \frac{\theta \log |\mathbf{C}|}{|\mathbf{C}|} + \frac{\lambda}{|\mathbf{A}|}$$

Proof. Let $n = |\mathbf{C}|$ and let S be a set of addresses of at most θ nodes in $\mathbf{C}$. For all $\ell \in \mathbf{L}$, we define $\mathcal{E}_1$ as the event that the address of at least one of the replicas of ℓ is in S and $\mathcal{E}_2$ as the event that at least one of the addresses in S is on the path to a replica of ℓ. Precisely,

$$\mathcal{E}_1 = \{\mathsf{server}(\ell) \in S\}, \qquad \mathcal{E}_2 = \{S \cap \mathsf{route}(\mathsf{fe}(\ell), \mathsf{server}(\ell)) \neq \emptyset\}$$

For Chord, we then have that,

$$\Pr[\,\mathsf{server}_{\chi_{\mathbf{C}}}(\ell) \in \mathsf{Vis}_{\chi_{\mathbf{C}}}(\mathsf{fe}(\ell), S)\,] = \Pr[\,\mathcal{E}_1 \vee \mathcal{E}_2\,] \leq \Pr[\,\mathcal{E}_1\,] + \Pr[\,\mathcal{E}_2\,], \tag{3.4}$$

We first bound the probability of $\mathcal{E}_1$,

$$\Pr[\,\mathcal{E}_1\,] = \Pr[\,\mathsf{server}(\ell) \in S\,] = \Pr\left[\,H_2(\ell) \in \cup_{N:H_1(N)\in S} \mathsf{arc}_{\chi_{\mathbf{C}}}(N)\,\right] \leq \frac{\lambda}{|\mathbf{A}|} \tag{3.5}$$

We now bound $\mathcal{E}_2$. By the union bound and the law of total probability, we have that,

$$\begin{aligned}
\Pr[\,\mathcal{E}_2\,] &= \Pr[\,S \cap \mathsf{route}(\mathsf{fe}(\ell), \mathsf{server}(\ell)) \neq \emptyset\,] \\
&\leq \sum_{N \in S} \Pr[\,N \in \mathsf{route}(\mathsf{fe}(\ell), \mathsf{server}(\ell))\,] \\
&= \sum_{N \in S} \sum_{r \in \mathbf{C}} \Pr[\,N \in \mathsf{route}(\mathsf{fe}(\ell), r) \mid \mathsf{server}(\ell) = r\,] \cdot \Pr[\,\mathsf{server}(\ell) = r\,] \qquad (3.6)
\end{aligned}$$

But note that,

$$\Pr[\,N \in \mathsf{route}(\mathsf{fe}(\ell), r) \mid \mathsf{server}(\ell) = r\,] = \frac{\log n}{n}$$

which follows from the fact that path lengths in Chord are at most $\log n$. Substituting this in Eq. (4.7) we get,

$$\begin{aligned}
\Pr[\mathcal{E}_2] &\leq \sum_{N \in S} \sum_{r \in \mathbf{C}} \frac{\log n}{n} \cdot \Pr[\mathsf{server}(\ell) = r] \\
&= \sum_{N \in S} \frac{\log n}{n} \sum_{r \in \mathbf{C}} \Pr[\mathsf{server}(\ell) = r] \\
&= \sum_{N \in S} \frac{\log n}{n} \\
&= \frac{\theta \log n}{n}
\end{aligned} \tag{3.7}$$

Finally, the Theorem follows by plugging Eqs. (4.6) and (4.8) into Eq. (4.5).

■

Corollary 3.5.6. *Let* $\mathbf{C}$ *be a set of active nodes. For all* $\theta \leq |\mathbf{C}|/(e \log |\mathbf{C}|)$*, Chord is* $(\varepsilon, \delta, \theta)$*-balanced for*

$$\varepsilon = \frac{\theta}{|\mathbf{C}|}\left(\log |\mathbf{C}| + 6 \log\left(\frac{|\mathbf{C}|}{\theta}\right)\right) \quad \text{and} \quad \delta = \frac{1}{|\mathbf{C}|^2} + (e^{-\sqrt{|\mathbf{C}|}} \cdot \log |\mathbf{C}|)$$

Proof. From Corollary 3.5.4, we know that for $\theta \leq |\mathbf{C}|/e$,

$$\Pr[\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \lambda] \geq 1 - \delta \quad \text{for} \quad \lambda = \frac{6|\mathbf{A}|\theta}{|\mathbf{C}|} \log \frac{|\mathbf{C}|}{\theta}$$

and δ as stated above in corollary statement. Therefore, from Theorem 3.5.5, we conclude that for $\theta \leq |\mathbf{C}|/e$,

$$\Pr[(H_1, \mathbf{C}) \text{ is } (\varepsilon, \theta)\text{-balanced}] \geq 1 - \delta \quad \text{for} \quad \varepsilon = \frac{\theta \log |\mathbf{C}|}{|\mathbf{C}|} + \frac{\lambda}{|\mathbf{A}|}$$

Substituting the value of λ in last equation, we conclude the proof.

■

Remark. It follows from Corollary 3.5.6 that

$$\varepsilon = O\left(\frac{\theta}{|\mathbf{C}|} \log |\mathbf{C}|\right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. Note that assigning labels uniformly at random to nodes would achieve $\varepsilon = \theta/|\mathbf{C}|$ so Chord balances data fairly well. Note that balance of Chord is only $\log |\mathbf{C}|$ factor away from optimal balance which is very good given that the optimal balance is achieved with no routing at all.

3.5.3 The Security of our Chord-based EDHT

In the following Corollary we formally state the security of the standard scheme when its underlying DHT is instantiated with Chord.

Corollary 3.5.7. *If* $|\mathbf{L}| = \Theta(2^k)$, $|I| \leq |\mathbf{C}|/(e \log |\mathbf{C}|)$, *and if* EDHT *is instantiated with Chord, then it is* $\mathcal{L}_\varepsilon$*-secure with probability at least* $1 - 1/|\mathbf{C}|^2 - (e^{-\sqrt{|\mathbf{C}|}} \cdot \log |\mathbf{C}|) - \mathsf{negl}(k)$ *in the random oracle model, where*

$$\varepsilon = \frac{\theta}{|\mathbf{C}|}\left(\log |\mathbf{C}| + 6 \log\left(\frac{|\mathbf{C}|}{\theta}\right)\right).$$

Proof. The corollary follows from Theorem 3.4.2, Theorem 3.5.6 and the fact that Chord has non-committing allocations when H_2 is modeled as a random oracle. Note that during the simulation, the probability that $\mathcal{A}$ queries H_2 on at least one of the strings $t_1, \ldots, t_q$ is at most $\mathsf{poly}(k)/|\mathbf{L}|$. This is because $\mathcal{A}$ is polynomially-bounded so it can make at most $\mathsf{poly}(k)$ queries to H_2. And since for all i, $t_i = f(\ell_i)$, where f is a random function, the probability that $\mathcal{A}$ queries H_2 on at least one of $t_1, \ldots, t_q$ is at most $\mathsf{poly}(k)/|\mathbf{L}|$. And since $|\mathbf{L}| = \Theta(2^k)$, this probability is negligible in k.

■

From the discussion of Theorem 3.5.6, we know that,

$$\varepsilon = O\left(\frac{|I|}{|\mathbf{C}|} \log |\mathbf{C}|\right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. Setting $|I| = |\mathbf{C}|/(\alpha \log |\mathbf{C}|)$, for some $\alpha \geq e$, we have $\varepsilon = O(1/\alpha)$. Recall that, on each query, the leakage function leaks the query equality with probability at most ε. So, intuitively, this means that if an α fraction of nodes are corrupted then, the adversary can expect to learn the operation equality of an $O(1/\alpha)$ fraction of client queries. Note that this confirms the intuition that distributing an STE scheme suppresses its leakage.

3.6 Encrypted Distributed Hash Tables in the Transient Setting

In this section we define the security of transient EDHTs and analyze the security of the standard construction in this setting.

3.6.1 Syntax and Security Definitions

Syntax. A transient EDHT is a collection of nine algorithms $\mathsf{EDHT}^+ = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get}, \mathsf{Leave}, \mathsf{Join})$. The first seven algorithms are the same as the perpetual setting. The eighth is an algorithm Leave executed by an existing node in network when it wishes to leave the network, whereas the ninth is an algorithm Join executed by a node willing to join the network. Both of them take nothing as input and output nothing but either halt the Daemon algorithm or make changes to the routing tables.

Functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{DHT}^+}$

$\mathcal{F}^{\mathcal{L}}_{\mathsf{DHT}^+}$ stores a dictionary DX initialized to empty, a set $\mathbf{C} \subseteq \mathbf{N}$ of active nodes, and a set $I \subseteq \mathbf{N}$ of corrupted nodes. It proceeds as follows, running with client $\mathcal{C}$, n active nodes in $\mathbf{C}$ and a simulator Sim:

- $\mathsf{Put}(\ell, v)$: Upon receiving a label/value pair (ℓ, v) from client $\mathcal{C}$, it sets $\mathsf{DX}[\ell] := v$, and sends the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{put}, \ell, v))$ to the simulator Sim.
- $\mathsf{Get}(\ell)$: Upon receiving a label ℓ from client C, it returns $\mathsf{DX}[\ell]$ to the client $\mathcal{C}$ and the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{get}, \ell, \perp))$ to the simulator Sim.
- $\mathsf{Leave}(N)$: Upon receiving $N \in \mathbf{C}$, it returns the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{leave}, N))$ to the simulator Sim and updates its set $\mathbf{C}$.
- $\mathsf{Join}(N)$: Upon receiving $N \in \mathbf{N} \setminus \mathbf{C}$, it returns the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{join}, N))$ to the simulator Sim and updates its set $\mathbf{C}$.

Figure 3.4: $\mathcal{F}^{\mathcal{L}}_{\mathsf{DHT}^+}$: The DHT^+ functionality parameterized with leakage function $\mathcal{L}$.

We assume in this work that when a node leaves the network, all the pairs stored at that node are "re-put" in the network and when a node joins the network all the pairs currently in the network are "re-put". We note that this is not the most efficient way to handle leaves and joins but in this work our focus is on security rather than efficiency and this strategy has the worst possible leakage.

Security. The real and ideal experiments are the same as in Section 3.4 with the following differences. First, the trusted party $\mathcal{T}$ runs $\mathsf{Overlay}$ and Alloc with $|\mathbf{N}|$ nodes; second, the environment selects and activates a subset a set $\mathbf{C} \subseteq \mathbf{N}$ of nodes in the beginning; third, the environment also sends Leave and Join operations adaptively along with Get and Put operations to nodes; and fourth, the ideal functionality of Figure 4.1 is replaced with the ideal functionality described in Figure 3.4.

3.6.2 The Standard EDHT in the Transient Setting

In the transient setting, the standard scheme is composed of nine algorithms $\mathsf{EDHT}^+ = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get}, \mathsf{Leave}, \mathsf{Join})$. The first seven algorithms are exactly the same as the standard EDHT scheme in the perpetual setting (See Figure 3.2). The Leave algorithm simply calls $\mathsf{DHT}^+.\mathsf{Leave}$ while the Join algorithm calls $\mathsf{DHT}^+.\mathsf{Join}$. We now turn to describing the leakage of this scheme. We start with a description of the leakage for join and leave operations and then discuss the leakage for put and get operations.

Join and leave leakage. Roughly speaking, during the execution of the scheme, the adversary sees leakage on label/value pairs that are either stored at corrupted nodes or routed through corrupted nodes. Now, when a join or leave operation occurs, label/value pairs are moved throughout the network (e.g., during a leave, the leaving node's pairs are redistributed to other nodes). At this point, the adversary could get new leakage about pairs that it had not seen before the leave/join operation. For example, this would occur if a previously unseen label/value pair (i.e., that was stored on the leaving node) gets routed through a corrupted node during the re-distribution.

To simulate a leave/join operation correctly, the simulator will have to correctly simulate the re-distribution of pairs including of pairs it has not seen yet. But at this stage, it does not even know how many such pairs exist. This is because it does not get executed on put operations for labels not stored or routed by corrupted nodes. To overcome this, we reveal to the simulator how many of these pairs exist through the leakage function.

This, however, affects the get and put leakages for these pairs: now that the pairs have been re-distributed to (or routed through) a corrupted node the adversary will receive get and put leakages on these pairs. There is a technical challenge here, which is that we do not know how to simulate *only* the pairs that are re-distributed to (or routed through) corrupted nodes, so to address this we additionally reveal to the simulator the leakage of all the previously unseen pairs. It is not clear if this is strictly necessary and it could be that the scheme achieves a "tighter" leakage function. Note that this does not affect new pairs, i.e., pairs that are added after the leave/join operation (until another leave/join operation occurs).

Note that by revealing the number κ of previously unseen pairs, one can compute the total number of put operations up to the last leave/join operation. We denote this value by τ and make it explicit in the leakage function for ease of exposition.

The leakage profile. We are now ready to formally describe the leakage profile achieved by the standard scheme in the transient setting.

- $\mathcal{L}_\varepsilon\Big(\mathsf{DX}, \Big\{(\mathsf{op}, \ell, v), (\mathsf{op}, N)\Big\}\Big)$:
  1. if $\mathsf{op} = \mathtt{get} \vee \mathtt{put}$ and ℓ has never been seen
     (a) sample and store $b_\ell \leftarrow \mathsf{Ber}(\varepsilon)$
  2. if $b_\ell = 1$
     (a) if $\mathsf{op} = \mathtt{put}$ output $(\mathtt{put}, \mathsf{opeq}(\ell))$
     (b) else if $\mathsf{op} = \mathtt{get}$ output $(\mathtt{get}, \mathsf{opeq}(\ell))$
  3. else if $b_\ell = 0$
     (a) Increment κ if $\mathsf{op} = \mathtt{put}$ and ℓ has never been seen
     (b) output $\perp$
  4. Increment τ
  5. if $\mathsf{op} = \mathtt{leave} \vee \mathtt{join}$
     (a) output $(\mathsf{op}, N, \kappa, \tau)$
     (b) set $b_\ell = 1$ for all the put labels that have been seen in the past
     (c) reset κ to 0

We now show that EDHT^+ is $\mathcal{L}_\varepsilon$-secure in the transient setting with probability negligibly close to $1 - \delta$ when its underlying transient DHT is $(\varepsilon, \delta, \theta)$-balanced and is non-committing.

Theorem 3.6.1. *If $|I| \leq \theta$ and DHT^+ is $(\varepsilon, \delta, \theta)$-balanced, has non-committing allocations and has label-independent allocation and front-end distributions, then EDHT^+ is $\mathcal{L}_\varepsilon$-secure with probability at least $1 - \delta - \mathsf{negl}(k)$.*

Proof. Consider the simulator Sim that works as follows. Given a set of corrupted nodes $I \subseteq \mathbf{N}$, and a set of active nodes $\mathbf{C} \subseteq \mathbf{N}$, it computes $\omega \leftarrow \mathsf{DHT}^+.\mathsf{Overlay}(n)$, $\psi \leftarrow \mathsf{DHT}^+.\mathsf{Alloc}(n, \omega, \rho)$, $\phi \leftarrow \mathsf{DHT}^+.\mathsf{FrontEnd}(\omega, n)$, initializes n nodes $N_1, \ldots, N_n$ in $\mathbf{C}$, simulates the adversary $\mathcal{A}$ with I as input, and generates a symmetric key $K \leftarrow \mathsf{SKE.Gen}(1^k)$. It then sets $I' = \mathbf{C} \cap I$. In the following, let $B = \mathsf{Vis}_{\omega,\mathbf{C}}(\mathsf{fe}_\phi(\ell), I')$, $G = \mathbf{A} \setminus B$, and $p' = \Pr[\,\psi(\ell) \in B\,]$, all of which are unique since we assume label independent allocations and label independent front-end nodes. If $p' > \varepsilon$, the simulator aborts, otherwise it continues. The simulator also initializes two empty multimaps MM and MM'.

When a leave/join operation is executed, the simulator receives from $\mathcal{F}_{\mathsf{DHT}^+}$ the leakage

$$\lambda \in \left\{ \left(\mathtt{leave}, N, \kappa, \tau \right), \left(\mathtt{join}, N, \kappa, \tau \right) \right\}.$$

For each $j \in [\kappa]$, it sets $t_j \xleftarrow{\$} \{0,1\}^d$ and $e_j \leftarrow \mathsf{SKE.Enc}(K, 0)$, samples an address $a \leftarrow \Delta(G)$, programs ψ to map t to a, computes $N' \leftarrow \mathsf{server}(t_j)$, and adds (t_j, e_j) to $\mathsf{MM}[N']$. It then sets $\mathsf{MM}'[\tau' \| \tau] = \{t_1, \ldots, t_\kappa\}$, where τ' is the time of the last leave/join operation. It also sets $b'_i = 1$ for all the put labels that have been seen in the past. Finally, if the operation is a leave operation, it updates $\mathbf{C} = \mathbf{C} \setminus \{N\}$, updates the routing tables to exclude N, and executes $\mathsf{DHT.Put}(t, v)$ on all the (t, v) pairs stored in $\mathsf{MM}[N]$, updating MM according to how pairs move.

If the operation is a join operation, it updates $\mathbf{C} = \mathbf{C} \cup \{N\}$, updates the routing tables to include N, and executes $\mathsf{DHT.Put}(t, v)$ on all the (t, v) pairs stored in MM for all the nodes, updating MM according to how pairs move. It finally, resets $\mathsf{MM}[N]$ to $\perp$, $I' = I \cap \mathbf{C}$, $B = \mathsf{Vis}_{\omega,\mathbf{C}}(\mathsf{fe}(\ell), I')$, $G = \mathbf{A} \setminus B$, and computes $p' = \Pr[\,\psi(\ell) \in B\,]$. If $p' > \varepsilon$, it aborts and exits, otherwise it continues.

When a put/get operation is executed, the simulator receives from $\mathcal{F}_{\mathsf{DHT}^+}$ leakage

$$\lambda \in \left\{ \left(\mathtt{put}, \mathsf{opeq}(\ell) \right), \left(\mathtt{get}, \mathsf{opeq}(\ell) \right), \perp \right\}.$$

If $\lambda = \perp$ then Sim does nothing. If $\lambda \neq \perp$, then Sim checks the operation equality to see if the label has been used in the past. If not, it samples and stores a bit

$$b' \leftarrow \mathsf{Ber}\left(\frac{p'}{\varepsilon} \right).$$

Note that, this is indeed a valid Bernoulli distribution since

$$\begin{aligned} p' &= \Pr[\,\psi(\ell) \in B\,] \\ &= \Pr[\,\mathsf{server}_{\omega,\psi}(\ell) \in \mathsf{Vis}_{\omega,\mathbf{C}}(\mathsf{fe}(\ell), I')\,] \\ &\leq \Pr[\,\mathsf{server}_{\omega,\psi}(\ell) \in \mathsf{Vis}_{\omega,\mathbf{C}}(\mathsf{fe}(\ell), I)\,] \\ &\leq \varepsilon, \end{aligned}$$

where the second equality follows from the definition of visible address, and the last two inequalities follows from $|I'| \leq |I| \leq \theta$ and $(\omega, \mathbf{C})$ being (ε, θ)-balanced.

It then sets $t \xleftarrow{\$} \{0,1\}^d$ and computes $e \leftarrow \mathsf{SKE.Enc}(K, 0)$. If $b' = 0$, and the operation is a put operation, it samples $a \leftarrow \Delta(G)$, otherwise, (if $b' = 1$ and irrespective of operation) it samples $a \leftarrow \Delta(B)$. In either case, it programs ψ to map t to a, computes $N' \leftarrow \mathsf{server}(t_j)$, adds (t_j, e_j) to $\mathsf{MM}[N']$, and executes $\mathsf{DHT.Put}(t, e)/\mathsf{DHT.Get}(t)$ depending on whether the operation was a put or a get.

If on the other hand, the label has been used in the past (as deduced from query equality), it retrieves the bit b' previously sampled. If $b' = 0$, it does nothing. If $b' = 1$, it sets t to the d-bit value previously used and $e \leftarrow \mathsf{SKE.Enc}(K, 0)$, and executes $\mathsf{DHT.Put}(t, e)/\mathsf{DHT.Get}(t)$. If $b' = \perp$, (this occurs for the labels which b was 0 initially but later leave/join occured), it sets $t = t'$, where $t' \xleftarrow{\$} \mathsf{MM}'[\tau \| \tau']$, such that $\mathsf{opeq}(\ell) \in [\tau, \tau']$, and then removes t' from $\mathsf{MM}'[\tau \| \tau']$. It finally computes $e \leftarrow \mathsf{SKE.Enc}(K, 0)$ and executes $\mathsf{DHT.Put}(t, e)/\mathsf{DHT.Get}(t)$.

Once all of the environment's operations are processed, the simulator returns whatever the adversary outputs.

It remains to show that the view of the adversary $\mathcal{A}$ during the simulation is indistinguishable from its view in a **Real** experiment. We do this using a sequence of games.

Game_0 : is the same as a $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ experiment.

Game_1 : is the same as Game_0 except that the encryption of the value v during a Put is replaced by $\mathsf{SKE.Enc}(K_2, 0)$.

Game_2 : is the same as Game_1 except that output of the PRF F is replaced by a truly random string of d bits.

Game_3 : is the same as Game_2 except that for each operation op, if $\mathsf{op} \in \{(\mathtt{get}, \ell), (\mathtt{put}, \ell, v)\}$, we check if the label ℓ has been seen before. If not, we sample and store a bit $b_\ell \leftarrow \mathsf{Ber}(\varepsilon)$, else we set b_ℓ to the bit previously sampled for ℓ. If $b_\ell = 1$ and $\mathsf{op} = (\mathtt{put}, \ell, v)$, we replace the Put operation with $\mathsf{Sim}(\mathtt{put}, \mathsf{opeq}(\ell))$ and if $\mathsf{op} = (\mathtt{get}, \ell)$ we replace the Get operation with $\mathsf{Sim}(\mathtt{get}, \mathsf{opeq}(\ell))$. If however $\mathsf{op} = (\mathtt{leave}, N)$, we replace the Leave operation with $\mathsf{Sim}(\mathtt{leave}, N, \kappa, \tau)$ and set $b_\ell = 1$ for all the put labels that have been seen in the past. Similarly if $\mathsf{op} = (\mathtt{join}, N)$, we replace the Join operation with $\mathsf{Sim}(\mathtt{join}, N, \kappa, \tau)$ and set $b_\ell = 1$ for all the put labels that have been seen in the past.

Game_1 is indistinguishable from Game_0, otherwise the encryption scheme is not semantically secure. Game_2 is indistinguishable from Game_1 because the outputs of pseudorandom functions are indistinguishable from random strings.

Let $(\omega, \mathbf{C})$ be the current overlay. Since DHT is $(\varepsilon, \delta, \theta)$-balanced, with probability at least $1 - \delta$, for all $\mathbf{C} \subseteq \mathbf{N}$, $(\omega, \mathbf{C})$ will be (ε, θ)-balanced. Furthermore, as shown in Theorem 3.4.2, if it is (ε, θ)-balanced then $p' \leq p$. It follows then that the simulator aborts with probability at most δ so for the rest of the proof, we argue indistinguishability assuming (ε, θ)-balanced overlays.

As in the proof of Theorem 3.4.2, we will consider the views of nodes in I' for each operation and show them to be indistinguishable across Game_2 and Game_3. We will denote this by $\mathbf{view_2}^i(I')$ and $\mathbf{view_3}^i(I')$ for Game_2 and Game_3 respectively.

Let $\mathbf{op}$ denote the sequence of operations generated by the environment. To prove the indistinguishability of views, we divide the operations in $\mathbf{op}$ into buckets where the bucket boundaries are the leave/join operations.

Now consider the first bucket. Since no leaves/joins have yet been simulated, b'_i can only be 0 or 1 but not $\perp$. Notice that for get and put operations in the bucket, when $b'_i = 1$, the simulator programs ψ in the same way as the simulator of Theorem 3.4.2. It does some extra bookkeeping in addition but that does not affect the view of the nodes in set I' for that operation. Moreover, for put operations when $b'_i = 0$, it only programs ψ to addresses not visible to I' and does nothing else which generates any extra view for nodes in I'. Therefore, using the same argument as in Theorem 3.4.2, we conclude that for get and put operations the views are indistinguishable.

Let op_i be the first leave/join operation (boundary of the first bucket) and let $t_1, \ldots, t_q$ be the distinct labels of put operations in first bucket. Now let A_r be the random variable denoting the allocation of $t_1, \ldots, t_q$ to addresses in $\mathbf{view_2}$. Then, using the law of total probability, we get

$$\begin{aligned}
&\Pr\left[\, \mathbf{view_2}^i(I') = \mathbf{v} \,\right] \\
&= \sum_{(\alpha_1,\ldots,\alpha_q)\in\mathbf{A}^q} \Pr\left[\, \mathbf{view_2}^i(I') = \mathbf{v} \mid A_r = (\alpha_1, \ldots, \alpha_q) \,\right] \cdot \Pr\left[\, A_r = (\alpha_1, \ldots, \alpha_q) \,\right] \qquad (3.8)
\end{aligned}$$

Similarly, let A_s be the random variable denoting the allocation of $t_1, \ldots, t_q$ to addresses in $\mathbf{view_3}$. Then,

$$\begin{aligned}
&\Pr\left[\, \mathbf{view_3}^i(I') = \mathbf{v} \,\right] \\
&= \sum_{(\alpha_1,\ldots,\alpha_q)\in\mathbf{A}^q} \Pr\left[\, \mathbf{view_3}^i(I') = \mathbf{v} \mid A_s = (\alpha_1, \ldots, \alpha_q) \,\right] \cdot \Pr\left[\, A_s = (\alpha_1, \ldots, \alpha_q) \,\right]
\end{aligned}$$

But conditioned on a fixed allocation $(\alpha_1, \ldots, \alpha_q) \in \mathbf{A}^q$ of labels, during leave/join operations, the views of the nodes in I' will be the same in both the games, since both of them will be re-distributing the same number of pairs using $\mathsf{DHT.Put}$. Therefore,

$$\Pr\left[\, \mathbf{view_2}^i(I') = \mathbf{v} \mid A_r = (\alpha_1, \ldots, \alpha_q) \,\right] = \Pr\left[\, \mathbf{view_3}^i(I') = \mathbf{v} \mid A_s = (\alpha_1, \ldots, \alpha_q) \,\right] \qquad (3.9)$$

Next we show that,

$$\Pr\left[\, A_r = (\alpha_1, \ldots, \alpha_q) \,\right] = \Pr\left[\, A_s = (\alpha_1, \ldots, \alpha_q) \,\right]$$

Notice that we can rewrite [3]

$$\Pr[\, A_r = (\alpha_1, \ldots, \alpha_q)\,] = \prod_{j \in [q]} \Pr[\, \psi(t_j) = \alpha_j\,] = \prod_{j \in [q]} \Pr[\, \psi(\ell) = \alpha_j\,]$$

where the last equality follows because ψ is a label-independent allocation function. The allocation in Game_3 is determined by the programmed ψ function. To avoid any confusion with the ψ function of Game_2, we denote by ψ_P, the programmed allocation function of Game_3. Then, we can rewrite,

$$\Pr[\, A_s = (\alpha_1, \ldots, \alpha_q)\,] = \prod_{j \in [q]} \Pr[\, \psi_P(t_j) = \alpha_j\,]$$

There are two subcases to consider. In the first case, $\alpha_j \in B$. Then,

$$\Pr[\, \psi_P(t_j) = \alpha_j\,] = \Pr\big[\, b_j = 1 \wedge b'_j = 1 \wedge a_j = \alpha_j\,\big]$$

where $a_j \leftarrow \Delta(B)$. Now,

$$\begin{aligned}\Pr\big[\, b_j = 1 \wedge b'_j = 1 \wedge a_j = \alpha_j\,\big] &= \varepsilon \cdot \frac{\Pr[\, \psi(\ell) \in B\,]}{\varepsilon} \cdot \frac{\Pr[\, \psi(\ell) = \alpha_j\,]}{\Pr[\, \psi(\ell) \in B\,]} \\ &= \Pr[\, \psi(\ell) = \alpha_j\,]\end{aligned}$$

In the second case, $\alpha_j \in \mathbf{A} \setminus B = G$. Then,

$$\Pr[\, \psi_P(t_j) = \alpha_j\,] = \Pr[\, \mathcal{E}_1\,] + \Pr[\, \mathcal{E}_2\,]$$

where

$$\begin{aligned}\Pr[\, \mathcal{E}_1\,] &= \Pr\big[\, b_j = 1 \wedge b'_j = 0 \wedge a_j = \alpha_j\,\big], \text{ and} \\ \Pr[\, \mathcal{E}_2\,] &= \Pr[\, b_j = 0 \wedge a_j = \alpha_j\,]\end{aligned}$$

such that $a_j \leftarrow \Delta(G)$. Then,

$$\begin{aligned}\Pr[\, \mathcal{E}_1\,] &= \Pr\big[\, b_j = 1 \wedge b'_j = 0 \wedge a_j = \alpha_j\,\big] \\ &= \varepsilon \cdot \left(1 - \frac{\Pr[\, \psi(\ell) \in B\,]}{\varepsilon}\right) \cdot \frac{\Pr[\, \psi(\ell) = \alpha_j\,]}{\Pr[\, \psi(\ell) \in G\,]}\ , \text{and}\end{aligned}$$

$$\begin{aligned}\Pr[\, \mathcal{E}_2\,] &= \Pr[\, b_j = 0 \wedge a_j = \alpha_j\,] \\ &= (1 - \varepsilon) \cdot \frac{\Pr[\, \psi(\ell) = \alpha_j\,]}{\Pr[\, \psi(\ell) \in G\,]}\end{aligned}$$

[3] there is an implicit assumption made here that for each label, its allocation to an address is independent of the previous allocations. However, the proof can be extended when no such assumption is made using the chain rule of probability.

Adding the two probabilites, we get,

$$\begin{aligned}\Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2] &= \frac{\Pr[\psi(\ell) = \alpha_j]}{\Pr[\psi(\ell) \in G]} \cdot \left(\varepsilon \cdot \left(1 - \frac{\Pr[\psi(\ell) \in B]}{\varepsilon}\right) + \left(1 - \varepsilon\right)\right) \\ &= \frac{\Pr[\psi(\ell) = \alpha_j]}{\Pr[\psi(\ell) \in G]} \cdot \left(1 - \Pr[\psi(\ell) \in B]\right) \\ &= \frac{\Pr[\psi(\ell) = \alpha_j]}{\Pr[\psi(\ell) \in G]} \cdot \Pr[\psi(\ell) \in G] \\ &= \Pr[\psi(\ell) = \alpha_j]\end{aligned}$$

Hence,

$$\Pr[A_r = (\alpha_1, \ldots, \alpha_q)] = \Pr[A_s = (\alpha_1, \ldots, \alpha_q)] \tag{3.10}$$

Therefore, by substituting Equations 3.9 and 3.10 in Equation 3.8, we conclude that at the first churn operation,

$$\Pr\left[\mathbf{view_2}^i(I') = \mathbf{v}\right] = \Pr\left[\mathbf{view_3}^i(I') = \mathbf{v}\right]$$

Moreover, since the allocation distribution before the churn operation is the same and both the games use the same DHT.Put to move the pairs, therefore, the new allocation distribution will also be the same. Hence using induction on each bucket, we prove that views will be indistinguishable for all the buckets. The proof follows by noticing that Game_3 is same as $\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k)$ experiment.

■

Efficiency. The time, round and communication complexities of leave and join operations of the standard scheme is transient setting are the same as the underlying DHT.

3.7 A Chord-Based EDHT in the Transient Setting

We now describe and analyze how Chord can work in a transient setting. The Chord paper does not precisely specify how joins and leaves should be handled. More precisely, it describes what the new destination nodes of the pairs should be but does not describe how exactly the pairs should get to their new destination nodes. Because of this, we describe here a simple approach based on "re-hashing". We note that this is not the most efficient way to handle leaves and joins but it is correct and our focus is on security rather than efficiency.

Leaves and joins in Chord. When a new node $N \in \mathbf{N} \setminus \mathbf{C}$ joins the network, it is first assigned an address $H_1(N) \in \mathbf{A}$. Then, the routing tables of all the other nodes are updated. Finally, all the label/value pairs stored at $\mathsf{succ}_{\chi_\mathbf{C}}(H_1(N))$ are re-hashed and stored at their new destination (if necessary). When a node $N \in \mathbf{C}$ leaves, the routing tables of all the other nodes are updated and all the label/value pairs stored at N are moved to the $\mathsf{succ}_{\chi_\mathbf{C}}(H_1(N))$.

3.7.1 Analysis of Chord's Stability

Recall from the security analysis of the Chord-based EDHT that its leakage was $\mathcal{L}_\varepsilon$, where ε is a function of the upper bound on $\mathsf{maxareas}$ and where the simulation error δ is a function of the probability of that bound.

In perpetual setting there is a single configuration corresponding to fixed set of active nodes. However, in transient setting there are multiple configurations – every time a node leaves/joins, the configuration changes. Therefore, in transient setting, the parameters ε and δ are functions of bounds on $\mathsf{maxareas}$ and their probabilities of each possible configuration.

We describe here, at a high level, two strategies for computing these parameters, with tradeoffs between quality of simulation and running time efficiency. The first strategy is efficient but has a δ which is $1/\mathsf{poly}(|\mathbf{N}|)$ while the second has expensive setup but improves δ to $\mathsf{negl}(k)$.

Approach #1. In this approach we upper bound the $\mathsf{maxareas}$ in the configuration $\chi_{\mathbf{C}}$ via the $\mathsf{maxareas}$ in the configuration $\chi_{\mathbf{N}}$. The approach relies on two main observations. The first is that any configuration $\chi_{\mathbf{C}}$ can be expressed as $\chi_{\mathbf{N}} \setminus \chi_{\mathbf{N}\setminus\mathbf{C}}$ which, intuitively, means we can recover $\chi_{\mathbf{C}}$ by starting with $\chi_{\mathbf{N}}$ (which includes every node in the name space) and removing the nodes $\mathbf{N} \setminus \mathbf{C}$. The second observation is that if we start with a given configuration $\chi_{\mathbf{C}}$ and remove a node N, then N's area becomes visible to some other (currently active) node.

But how exactly can we use these observations to bound the maximum area in $\chi_{\mathbf{C}}$ using the maximum area in $\chi_{\mathbf{N}}$? We start with $\chi_{\mathbf{N}}$ and remove the nodes in $\mathbf{N} \setminus \mathbf{C}$; but for each node N that is removed, we assume the worst-case and assign N's area to one of the θ nodes with largest arcs. The resulting area will be an upper bound on the true maximum area. More formally, we have that $\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + |\mathbf{N}| - |\mathbf{C}|)$.

For our purposes, we will need to show that this bound holds for all large enough $\mathbf{C}$'s so the next step will be to prove that if $|\mathbf{N}| - |\mathbf{C}| \leq d$, then for all $\mathbf{C}$ such that $|\mathbf{C}| \geq |\mathbf{N}| - d$, $\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + |\mathbf{N}| - |\mathbf{C}|)$ is upper bounded by $\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d)$. But we can bound the latter using Corollary 3.5.4 with probability at least $1 - O(1/|\mathbf{N}|^2)$.

Approach #2. The limitation of the previous approach is that the bound only holds with probability $1 - O(1/|\mathbf{N}|^2)$ which leads to a $O(1/|\mathbf{N}|^2)$ error probability for the simulation. Using our second approach, however, we will reduce the error probability to be negligible.

We do this by using a new overlay algorithm $\overline{\mathsf{Overlay}}$ that works as follows. It runs the old $\mathsf{Overlay}$ algorithm $r = O(k/\log|\mathbf{N}|)$ times in the hope of sampling an overlay parameter $\omega = H_1$ such that $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1)$ is small. We show in Lemma 3.7.4 that $\overline{\mathsf{Overlay}}$ will find such an H_1 with overwhelming probability in k.

Using $\overline{\mathsf{Overlay}}$, one can find, with overwhelming probability, an overlay with a small $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1)$. This, in turn, gives us a bound on $\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d)$ with overwhelming probability (Corollary 3.5.4) which yields a simulation with negligible error probability. As we will see, the main limitation of this approach is that $\overline{\mathsf{Overlay}}$ runs in time $O(k|\mathbf{N}|)$ as opposed to $\mathsf{Overlay}$ which runs in $O(1)$ time.

We however show experimentally, that the probability of sampling a good hash function in a *single* trial is very high (and seems to grow exponentially). Therefore, for practical purposes, it is most likely enough to use Overlay instead of $\overline{\mathsf{Overlay}}$.

3.7.2 Approach #1: High Probability Simulation Success

Here, we analyze our first strategy. We start by proving a Lemma that bounds the maximum areas of all the configurations $\chi_{\mathbf{C}}$ with large enough $\mathbf{C}$.

Lemma 3.7.1. *If $\chi_{\mathbf{N}} = (H_1, \mathbf{N})$ is a configuration such that*

$$\Pr\left[\,\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d) \leq \alpha\,\right] \geq 1 - \beta,$$

then,

$$\Pr\left[\bigwedge_{\mathbf{C}\subseteq\mathbf{N}:|\mathbf{C}|\geq|\mathbf{N}|-d} \mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \alpha\right] \geq 1 - \beta.$$

Proof. We prove this by contradiction. Suppose that there exists a set of active nodes $\mathbf{C}^*$ and a subset of nodes $S \subseteq \mathbf{C}^*$ such that $|\mathbf{C}^*| \geq |\mathbf{N}| - d$ and that $|S| = \theta$ for which $\mathsf{area}(\chi_{\mathbf{C}^*}, S) > \alpha$. We then show that there exists a set of nodes $D \subseteq \mathbf{N}$ of size $\theta + d$ such that $\mathsf{area}(\chi_{\mathbf{N}}, D) > \alpha$.

Consider the set $D = S \cup \mathbf{N} \setminus \mathbf{C}^*$ and note that

$$\mathsf{area}(\chi_{\mathbf{N}}, D) = \mathsf{area}(\chi_{\mathbf{N}}, S \cup \mathbf{N} \setminus \mathbf{C}^*) = \mathsf{area}(\chi_{\mathbf{N}}, S) + \mathsf{area}(\chi_{\mathbf{N}}, \mathbf{N} \setminus \mathbf{C}^*) \tag{3.11}$$

We know that for some subset $Z \subseteq \mathbf{N} \setminus \mathbf{C}^*$, the following holds:

$$\begin{aligned}\mathsf{area}(\chi_{\mathbf{C}^*}, S) &= \mathsf{area}(\chi_{\mathbf{N}}, S) + \mathsf{area}(\chi_{\mathbf{N}}, Z) \\ &\leq \mathsf{area}(\chi_{\mathbf{N}}, S) + \mathsf{area}(\chi_{\mathbf{N}}, \mathbf{N} \setminus \mathbf{C}^*),\end{aligned} \tag{3.12}$$

where the equality holds because when nodes in $\mathbf{N} \setminus \mathbf{C}^*$ are removed from $\chi_{\mathbf{N}}$, their areas might become visible to nodes in S, and the inequality holds because $Z \subseteq \mathbf{N} \setminus \mathbf{C}^*$. From Equations 3.11 and 3.12, we conclude that

$$\mathsf{area}(\chi_{\mathbf{N}}, D) \geq \mathsf{area}(\chi_{\mathbf{C}^*}, S) > \alpha$$

where the last inequality follows from our assumption. This, however, is a contradiction.

Since $\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d) \leq \alpha$ implies the that for all $\mathbf{C} \subseteq \mathbf{N}$ such that $|\mathbf{C}| \geq \mathbf{N} - d$, $\mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \alpha$, if the former occurs with probability at least $1 - \beta$ then so does the latter.

■

Stability of Chord. We now turn to proving the stability of Chord.

Theorem 3.7.2. *For all $\theta \leq |\mathbf{N}|/(e \log |\mathbf{N}|) - d$, transient Chord is $(\varepsilon, \delta, \theta)$-stable for*

$$\varepsilon = \frac{(\theta + d)}{|\mathbf{N}|}\left(\log|\mathbf{N}| + 6\log\left(\frac{|\mathbf{N}|}{(\theta + d)}\right)\right) \quad \text{and} \quad \delta = \frac{1}{|\mathbf{N}|^2} + (e^{-\sqrt{|\mathbf{N}|}} \cdot \log|\mathbf{N}|)$$

Proof. From Lemma 3.7.1, we know that if

$$\Pr\left[\,\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d) \leq \alpha\,\right] \geq 1 - \beta$$

then

$$\Pr\left[\bigwedge_{\mathbf{C}\subseteq\mathbf{N}:|\mathbf{C}|\geq\mathbf{N}-d} \mathsf{maxareas}(\chi_{\mathbf{C}}, \theta) \leq \alpha\right] \geq 1 - \beta.$$

But from Corollary 3.5.4, we have that for $\theta + d \leq |\mathbf{N}|/e$,

$$\alpha = \frac{6|\mathbf{A}|(\theta + d)}{|\mathbf{N}|} \log\left(\frac{|\mathbf{N}|}{(\theta + d)}\right), \qquad \beta = \frac{1}{|\mathbf{N}|^2} + (e^{-\sqrt{|\mathbf{N}|}} \cdot \log|\mathbf{N}|)$$

Finally by applying Theorem 3.5.5, we conclude that for $\theta + d \leq |\mathbf{N}|/e$,

$$\Pr\left[\bigwedge_{\mathbf{C}\subseteq\mathbf{N}:|\mathbf{C}|\geq\mathbf{N}-d} (H_1, \mathbf{C}) \text{ is } (\varepsilon, \theta)\text{-balanced}\right] \geq 1 - \delta,$$

where ε and δ are as defined in theorem statement.

■

Security of the Chord-based EDHT. In the following Corollary we formally state the security of the standard EDHT when its underlying DHT is instantiated with transient Chord.

Corollary 3.7.3. *If $|\mathbf{L}| = \Theta(2^k)$, $|I| \leq |\mathbf{N}|/(e \log|\mathbf{N}|) - d$, and if EDHT^+ is instantiated with Chord, then it is $\mathcal{L}_\varepsilon$-secure with probability at least $1 - 1/|\mathbf{N}|^2 - (e^{-\sqrt{|\mathbf{N}|}} \cdot \log|\mathbf{N}|) - \mathsf{negl}(k)$ in the random oracle model, where*

$$\varepsilon = \frac{(|I| + d)}{|\mathbf{N}|}\left(\log|\mathbf{N}| + 6\log\left(\frac{|\mathbf{N}|}{|I| + d}\right)\right)$$

The corollary follows from Theorems 3.6.1 and 3.7.2 and the fact that Chord has non-committing allocations when H_2 is modeled as a random oracle. The proof is the same as the proof of Corollary 3.5.7.

Practical considerations. Similar to the discussion following Corollary 3.5.7 in the perpetual setting, if we set $|I| + d \leq |\mathbf{N}|/(\alpha \log|\mathbf{N}|)$, where $\alpha \geq e$, then, in expectation, the adversary will learn the query equality leakage of an $O(1/\alpha)$ fraction of the queries executed between any two churn operations. One thing to notice here is that the inequality $|I| + d \leq |\mathbf{N}|/(e \log|\mathbf{N}|)$ implies that $|I| + |\mathbf{N}| - |\mathbf{C}| \leq (e \log|\mathbf{N}|)$ which, in turn, implies that $|\mathbf{C}| \geq ((e \log|\mathbf{N}| - 1)/e \log|\mathbf{N}|)|\mathbf{N}| + |I|$. Concretely, this means that at all times, the network must have at least $((e \log|\mathbf{N}| - 1)/e \log|\mathbf{N}|)|\mathbf{N}| + |I|$ nodes which bounds how many nodes can ever leave the system.

3.7.3 Approach #2: Achieving an Overwhelming Bound on Simulation Success

We now analyze our second strategy which yields an overwhelming bound on the simulation's success probability. As discussed above, we do this by using a new overlay algorithm $\overline{\mathsf{Overlay}}$, which amplifies the probability that $\mathsf{Overlay}$ outputs a good hash function. $\overline{\mathsf{Overlay}}$ takes as input an integer $n \geq 1$ and the security parameter k and chooses a hash function by executing $H_1 \leftarrow \mathsf{DHT.Overlay}(n)$ and checking whether $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1) \leq (1 + c_1)|\mathbf{A}| \log |\mathbf{N}|/|\mathbf{N}|$, where $\chi_{\mathbf{N}} = (H_1, \mathbf{N})$. If so, it outputs H_1 otherwise it retries for a maximum $r = k/(c_1 \log |\mathbf{N}|)$ times, in which case it fails.

We now turn to analyzing the stability of Chord where the $\mathsf{Overlay}$ algorithm has been replaced by $\overline{\mathsf{Overlay}}$. Since most of the details are similar to what was in the Approach #1, we keep the description high level.

We start by showing that with overwhelming probability $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1)$ is small. We then use it to show that with overwhelming probability $\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta + d)$ is also small.

Lemma 3.7.4. *Let H_1 be the hash function output by $\overline{\mathsf{Overlay}}$ and let $\chi_{\mathbf{N}} = (H_1, \mathbf{N})$ be the configuration induced by H_1. Then,*

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{N}}, 1) \leq \frac{(1 + c_1)|\mathbf{A}| \log |\mathbf{N}|}{|\mathbf{N}|}\right] \geq 1 - \mathsf{negl}(k),$$

where the probability is over the coins of $\overline{\mathsf{Overlay}}$.

Proof. We call an H_1 bad if $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1)$ is greater than $(1 + c_1)|\mathbf{A}| \log |\mathbf{N}|/|\mathbf{N}|$. Let $\mathcal{E}_i$ be the event that a bad H_1 is sampled in the i-th trial. Then the failure probability of $\overline{\mathsf{Overlay}}$ (i.e., of getting a bad H_1 at the end of $\overline{\mathsf{Overlay}}$) is:

$$\Pr\left[\bigwedge_{1 \leq i \leq r} \mathcal{E}_i\right] \leq \frac{1}{|\mathbf{N}|^{c_1 r}} = \frac{1}{e^{c_1 r \log |\mathbf{N}|}} = \frac{1}{e^k} = \mathsf{negl}(k),$$

where the first inequality follows from Lemma 3.5.2 and from the fact that the $\mathcal{E}_i$'s are independent, and the last equality follows by setting $r = k/(c_1 \log |\mathbf{N}|)$.

■

Corollary 3.7.5. *Let H_1 be the hash function output by $\overline{\mathsf{Overlay}}$ and let $\chi_{\mathbf{N}} = (H_1, \mathbf{N})$ be the configuration induced by H_1. If $|\mathbf{N}| = \Omega(k)$ and $\theta \leq |\mathbf{N}|/e$,*

$$\Pr\left[\mathsf{maxareas}(\chi_{\mathbf{N}}, \theta) \leq \frac{6|\mathbf{A}|\theta}{|\mathbf{N}|} \log \frac{|\mathbf{N}|}{\theta}\right] \geq 1 - \mathsf{negl}(k).$$

The proof is similar to the proof of Corollary 3.5.4. The difference is that the probability p_1 that $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1)$ is bounded by $(3|\mathbf{A}| \log |\mathbf{N}|)/|\mathbf{N}|$ is at most $\mathsf{negl}(k)$ (from Lemma 3.7.4). The Corollary follows by setting $p_1 = \mathsf{negl}(k)$ and $|\mathbf{N}| = \Omega(k)$.

Stability and security. We now turn to the stability and the security of the Chord-based EDHT.

Theorem 3.7.6. *If* $|\mathbf{N}| = \Omega(k)$ *and* $\theta \leq |\mathbf{N}|/(e \log |\mathbf{N}|) - d$*, transient Chord is* $(\varepsilon, \delta, \theta)$*-stable for*

$$\varepsilon = \frac{(\theta + d)}{|\mathbf{N}|}\left(\log |\mathbf{N}| + 6 \log\left(\frac{|\mathbf{N}|}{\theta + d}\right)\right) \quad \text{and} \quad \delta = \mathsf{negl}(k)$$

The proof is exactly same as the proof of Theorem 3.7.2 with the exception that that we use Corollary 3.7.5 to compute β instead of using Corollary 3.5.4.

Corollary 3.7.7. *If* $|\mathbf{L}| = \Theta(2^k)$*,* $|\mathbf{N}| = \Omega(k)$ *and* $|I| \leq |\mathbf{N}|/e - d$*, and if* EDHT^+ *is instantiated with Chord, then it is* $\mathcal{L}_\varepsilon$*-secure with probability at least* $1 - \mathsf{negl}(k)$ *in the random oracle model, where*

$$\varepsilon = \frac{(|I| + d)}{|\mathbf{N}|}\left(\log |\mathbf{N}| + 6 \log\left(\frac{|\mathbf{N}|}{|I| + d}\right)\right)$$

Efficiency of $\overline{\mathsf{Overlay}}$. Let $\alpha = ((1 + c_1)|\mathbf{A}| \log |\mathbf{N}|)/|\mathbf{N}|$. For each sampled hash function, $\overline{\mathsf{Overlay}}$ checks whether $\mathsf{maxareas}(\chi_{\mathbf{N}}, 1) \leq \alpha$. To do this, it computes $H_1(N)$ for all $N \in \mathbf{N}$, sorts all the $H_1(N)$'s to construct $\chi_{\mathbf{N}}$ and, for all $N \in \mathbf{N}$, checks if $\mathsf{area}(\chi_{\mathbf{N}}, N) \leq \alpha$. Sorting is $O(|\mathbf{N}| \log |\mathbf{N}|)$ while the remaining steps are $O(|\mathbf{N}|)$. Moreover, $\overline{\mathsf{Overlay}}$ takes a maximum of $k/(c_1 \log |\mathbf{N}|)$ samples so its total running time is $O(k|\mathbf{N}|)$.

Chapter 4

Encrypted Key-Value Stores

4.1 Introduction

A distributed key-value store (KVS) is a distributed storage system that stores label/value [1] pairs and supports get and put queries. KVSs provide one of the simplest data models but have become fundamental to modern systems due to their high performance, scalability and availability. For example, some of the largest social networks, e-commerce websites, cloud services and community forums depend on key-value stores for their storage needs. Prominent examples of distributed KVSs include Amazon's Dynamo [55], Facebook's Cassandra [82], LinkedIn's Voldemort [111], Redis [9], MemcacheDB [7] and Riak [113].

Distributed KVSs are closely related to distributed hash tables (DHT) and, in fact, most are built on top of a DHT. However, since DHTs do not necessarily guarantee fault-tolerance, KVSs use various techniques to achieve availability in the face of node failures. The simplest approach is to replicate each label/value pair on multiple nodes and to use a replica control protocol to guarantee some form of consistency.

End-to-end encryption in KVSs. As an increasing amount of data is being stored and managed by KVSs, their security has become an important problem. Encryption is often proposed as a solution, but encrypting data in transit and at rest and decrypting it before use is not enough since each decryption exposes the data and increases its likelihood of being stolen. A better way to protect data is to use end-to-end encryption where a data owner encrypts its data with its own secret key (that is never shared). End-to-end encryption guarantees that data is encrypted at all times—even in use—which ensures data confidentiality.

Our contributions. In this work, we formally study the use of end-to-end encryption in KVSs. In particular, we extend our framework from DHTs to KVSs. We formalize the goals of encryption in

[1] In this work we use the term *label* and reserve the term *key* to denote cryptographic keys.

KVSs by introducing the notion of an *encrypted key-value store* (EKVS) and propose formal syntax and security definitions for these objects. Similar to the standard EDHT scheme, we design an EKVS that stores label/value pairs (ℓ, v) as $(F_{K_1}(\ell), \mathsf{Enc}_{K_2}(v))$ in a standard/plaintext KVS, where F is a pseudo-random function and Enc is a symmetric encryption scheme. The underlying KVS will then replicate the encrypted pair, store the replicas on different storage nodes, handle routing, node failures and consistency. Throughout, we will refer to this approach as the *standard scheme* and we will use our framework to formally study its security properties. We make the following contributions:

- *formalizing KVSs*: we provide an abstraction of KVSs that enables us to isolate and analyze several important properties of standard/plaintext KVSs that impact the security of the standard EKVS. More precisely, we find that the way a KVS distributes its data and the extent to which it load balances have a direct effect on what information an adversary can infer about a client's queries.

- *distributed leakage analysis*: an EKVS can be viewed as a distributed version of an encrypted dictionary which is a fundamental building block in the design of sub-linear encrypted search algorithms (ESA). All sub-linear ESAs leak some information—whether they are built from property-preserving encryption, structured encryption or oblivious RAMs—so our goal is to identify and prove the leakage profile of the standard scheme. Leakage analysis in the distributed setting is particularly challenging because the underlying distributed system (in our case the underlying KVS) can create very subtle correlations between encrypted data items and queries. As we will see, replication makes this even more challenging. We consider two cases: the single-user case where the EKVS stores the datasets of multiple clients but each dataset can only be read and updated by its owner; and the multi-user case where each dataset can be read and updated by multiple users.

- *leakage in the multi-user case*: We show that in the multi-user setting, the standard scheme leaks the operation equality (i.e., if and when get and put operations are for the same label) over all operations; even operations that are not handled by corrupted nodes. [2] This may seem surprising since it is not clear a-priori why an adversary would learn anything about data that it never "sees".

- *leakage in the single-user case*: In the single-user scenario, we show that, if the standard scheme's underlying KVS achieves read your write (RYW) consistency, then it only leaks the operation equality over operations that are handled by corrupted nodes. This is particularly interesting as it suggests that stronger consistency guarantees improve the security of end-to-end encrypted KVSs.

- *comparison with DHTs*: As mentioned earlier, the main difference between a DHT and a KVS is that the latter replicate data on multiple nodes. To ensure a consistent view of this data,

[2] Note that the operation equality is a common leakage pattern in practical ESAs.

KVSs need to implement some consistency model. Achieving strong consistency, however, is very costly so almost all practical systems achieve weaker notions which cannot guarantee that a unique value will always be associated to a given label. In particular, the value that will be returned will depend on factors such as network delay, synchronization policy and the ordering of concurrent operations. Therefore, an adversary that controls one or more of these factors can affect the outputs of a KVS. It therefore becomes crucial to understand and analyze this correlation when considering the security of an encrypted KVS. In contrast, this is not needed in the case of encrypted DHTs since they do not maintain replicas and hence consistency is not an issue.

- *concrete instantiations*: We use our framework to study two concrete instantiations of the standard scheme. The first uses a KVS based on consistent-hashing with zero-hop routing whereas the second uses a KVS based on consistent hashing with multi-hop routing.

4.2 Related Work

Since we already discussed about encrypted search, structured encyrption, and distributed hash tables in the previous chapter, we only focus here on the related work on plaintext key-value stores and consistency guarantees.

Key-value stores. NoSQL databases were developed as an alternative to relational databases to satisfy the performance and scalability requirements of large Internet complanies. KVSs are the simplest kind of NoSQL databases. Even though such databases had already existed, they gained popularity when Amazon developed Dynamo [55], a KVS for its internal use. Since then many KVSs have been developed both in industry and academia. Most prominent ones are Facebook's Cassandra [82], Google's BigTable [51], LinkedIn's Voldemort [111], Redis [9], MemcacheDB [7] and Riak [113]. All of them are eventually consistent but some of them can be tuned to provide strong consistency [113, 9, 82]. There have also been efforts to develop KVSs with stronger consistency such as causal consistency [83, 84, 119, 31], and strong consistency [4, 2, 6, 10].

Consistency guarantees. The consistency guarantee of a distributed system specifies the set of acceptable responses that a read operation can output. There are multiple consistency guarantees studied in the literature, including *linearizability, sequential consistency, causal consistency and eventual consistency*. Though strong consistency notions like linearizability are desirable, the Consistency, Availability, and Partition tolerance (CAP) Theorem states that strong consistency and availability cannot be achieved simultaneously in the presence of network partitions. Therefore, many practical systems settle for weaker consistency guarantees like sequential consistency, causal consistency and eventual consistency. We note that all these weaker consistency guarantees—with the exception of eventual consistency—all satisfy what is known as "Read Your Writes" (RYW)

consistency which states that all the writes performed by a single client are visible to its subsequent reads. Many practical systems [9, 83, 84, 4, 2] guarantee RYW consistency.

4.3 Key-Value Stores

Here we extend the formal treatment of encrypted DHTs to key-value stores. A key-value store is a distributed storage system that provides a key-value interface and that guarantees *resiliency* against node failures. It does so by replicating label/value pairs on multiple nodes. Similar to DHTs, there are two kinds of KVSs: perpetual and transient. Perpetual KVSs are composed of a fixed set of nodes that are all known at setup time. Transient KVSs, on the other hand, are designed for settings where nodes are not known a-priori and can join and leave at any time. In this work, we study the security of pertpetual KVSs.

Since KVSs are closely related to DHTs, in the sense that both of them store and retrieve label/value pairs, the formalism for KVSs—its syntax, abstraction and security definition—looks very similar to that of the DHTs. However, unlike DHTs, since KVSs replicate data, we extend the framework to account for replication. For example, the server mapping which initially mapped a label to a single address in DHTs, now maps it to multiple addresses where each address is the address of the replica that stores the label. To avoid any confusion, we call the modified mapping the $\mathsf{replicas}$ mapping.

In this Section, instead of repeating text from previous chapter, we keep concepts high-level and only describe the modifications needed to adapt them for KVSs.

Perpetual KVSs. Similar to DHTs, we formalize KVSs as a collection of six algorithms $\mathsf{KVS} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$. All the algorithms, with the exception of Alloc, are same as before. Alloc along with ω and n, also takes as input an additional parameter $\rho \geq 1$ and outputs ψ same as before. Intuitively, ρ is the replication parameter and represents the number of nodes a label/value pair should be allocated to.

Abstracting KVSs. Similar to DHTs, we describe KVSs using a tuple of function families $(\mathsf{addr}, \mathsf{replicas}, \mathsf{route}, \mathsf{fe})$ that are all paramterized by a subset of $\{\omega, \psi, \phi\}$. These functions are defined as

$$\mathsf{addr}_\omega : \mathbf{N} \to \mathbf{A} \qquad \mathsf{replicas}_{\omega,\psi} : \mathbf{L} \to 2^{\mathbf{A}} \qquad \mathsf{route}_\omega : \mathbf{A} \times \mathbf{A} \to 2^{\mathbf{A}}, \qquad \mathsf{fe}_\phi : \mathbf{L} \to \mathbf{A}$$

where addr_ω, route_ω and fe_ϕ are same as before. Since KVSs replicate label/value pairs on multiple nodes for fault tolerance, we replace the server mapping by $\mathsf{replicas}$. It maps labels from a label space $\mathbf{L}$ to the set of addresses of ρ nodes that store it

Visible addresses. We extend the notion of visibility in a natural way. We say that an address $a \in \mathbf{A}$ is visible to a node N, if a label mapped to a is either stored at N or is routed by N. More

formally, for a fixed overlay $(\omega, \mathbf{C})$ and a fixed replication parameter ρ, an address $a \in \mathbf{A}$ is $\mathbf{s}$-visible to a node $N \in \mathbf{C}$ if for a label ℓ which ψ allocates to a, either: (1) $\mathsf{addr}_\omega(N) \in \mathsf{replicas}_{\omega,\psi}(\ell)$; or (2) $\mathsf{addr}_\omega(N) \in \mathsf{route}_\omega(s, \mathsf{replicas}_{\omega,\psi}(\ell))$.

Front-end and allocation distributions. The notions of front-end distribution captured by $\phi(\ell)$, allocation distribution captured by $\psi(\ell)$ remain the same as before. We also assume that both these distributions are labe independent in the sense that every label's front-end and allocation distribution is the same.

Non-committing allocations. As in the case of the EDHTs, the security of the standard EKVS requires the underlying KVS to have non-committing allocations and be balanced. A KVS is said to have non-committing allocations if for any label ℓ and an address a, its allocation parameter ψ can be programmed to map ℓ to a, i.e., $\psi(\ell)$ can be set to a. All the KVSs that we are aware have hash functions as their allocation parameters, which are programmable in the random oracle model.

Balanced overlays. Recall that this property was related to how well the DHT load balanced the label/value pairs it stored. Intuitively, it said that an overlay $(\omega, \mathbf{C})$ is balanced if for all labels ℓ, the probability that any set of θ nodes sees ℓ is not too large. For KVSs, the intuition remains the same. However, this should hold with any replication parameter ρ.

Definition 4.3.1 (Balanced overlays)**.** *Let $\omega \in \Omega$ be an overlay parameter, $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes, and $\rho \geq 1$ be a replication parameter. We say that an overlay $(\omega, \mathbf{C})$ is (ε, θ)-balanced if for all $\ell \in \mathbf{L}$, and for all $S \subseteq \mathbf{C}$ with $|S| = \theta$,*

$$\Pr\left[\, \mathsf{replicas}_{\omega,\psi}(\ell) \cap \mathsf{Vis}_{\omega,\mathbf{C},\rho}(\mathsf{fe}_\phi(\ell), S) \neq \emptyset \,\right] \leq \varepsilon,$$

where the probability is over the coins of Alloc *and* FrontEnd*, and where ε can depend on θ.*

Definition 4.3.2 (Balanced KVS)**.** *We say that a key-value store* $\mathsf{KVS} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ *is $(\varepsilon, \delta, \theta)$-balanced if for all $\mathbf{C} \subseteq \mathbf{N}$, the probability that an overlay $(\omega, \mathbf{C})$ is (ε, θ)-balanced is at least $1 - \delta$ over the coins of* Overlay *and where ε and δ can depend on $\mathbf{C}$ and θ.*

4.4 Encrypted Key-Value Stores

In this Section, we formally define encrypted key-value stores. An EKVS is an end-to-end encrypted distributed system that instantiates a replicated dictionary data structure.

4.4.1 Syntax and Security Definitions

Syntax. We formalize EKVSs as a collection of seven algorithms $\mathsf{EKVS} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$. The first algorithm Gen is executed by a client and takes as input a security parameter 1^k and outputs a secret key K. All the other algorithms have the same syntax as before (See Section 4.3), with the difference that Get and Put also take the secret key K as input.

Functionality $F_{\mathsf{KVS}}^{\mathcal{L}}$

$F_{\mathsf{KVS}}^{\mathcal{L}}$ stores a dictionary DX initialized to empty and proceeds as follows, running with client $\mathcal{C}$, n nodes $N_1, \ldots, N_n$ and a simulator Sim:

- $\mathsf{Put}(\ell, v)$: Upon receiving a label/value pair (ℓ, v) from client $\mathcal{C}$, it sets $\mathsf{DX}[\ell] := v$, and sends the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{put}, \ell, v))$ to the simulator Sim.
- $\mathsf{Get}(\ell)$: Upon receiving a label ℓ from client C, it returns $\mathsf{DX}[\ell]$ to the client $\mathcal{C}$ and the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{get}, \ell, \perp))$ to the simulator Sim.

Figure 4.1: $F_{\mathsf{KVS}}^{\mathcal{L}}$: The KVS functionality parameterized with leakage function $\mathcal{L}$.

Security. The definition is a leakage-based security definition and is exactly the same as before and is based on the real/ideal-world paradigm. We design two probabilistic experiments, **Real** and **Ideal**, where in the former parties interact as per the actual EKVS protocol, while in the latter, the parties interact with a trusted functionality shown in Figure 4.1. The protocol is secure if no environment can distinguish between the outputs of these two experiments.

Definition 4.4.1 ($\mathcal{L}$-security)**.** *We say that an encrypted key-value store* $\mathsf{EKVS} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ *is $\mathcal{L}$-secure, if for all* PPT *adversaries $\mathcal{A}$ and all* PPT *environments $\mathcal{Z}$, there exists a* PPT *simulator* Sim *such that for all $z \in \{0,1\}^*$,*

$$|\Pr[\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k) = 1] - \Pr[\mathbf{Ideal}_{\mathsf{Sim},\mathcal{Z}}(k) = 1]| \leq \mathsf{negl}(k).$$

Discussion on security definition. In the real/ideal-world paradigm, the security of a protocol is tied to its correctness. It is therefore important that our ideal functionality capture the correctness of the KVS as well. What this means is that the functionality should produce outputs that follow the same distribution as the outputs from a KVS. Unfortunately, in a setting with multiple clients sharing the data, even with the strongest consistency guarantees (e.g., linearizability), there are multiple possible responses for a read, and the one which the KVS actually outputs depends on behaviour of the network. Since the network behaviour is non-deterministic, the distribution over the possible outputs is also non-deterministic and hence the functionality cannot model the distribution over outputs correctly without modelling the network inside it.

However, if we restrict to a single client setting, RYW property ensures that a Get always outputs the latest value written to the KVS. Therefore the functionality $\mathcal{F}_{\mathsf{KVS}}$ models the correct distribution over the outputs: on a $\mathsf{Get}(\ell)$, it outputs the last value written to $\mathsf{DX}[\ell]$, and on a $\mathsf{Put}(\ell, v)$, it updates the $\mathsf{DX}[\ell]$ to v.

4.5 The Standard EKVS Scheme in the Single-User Setting

We now repeat the standard scheme from Chapter 3. The difference is that instead of using a DHT to store encrypted data, we now use a KVS to store encrypted data. This approach relies on simple cryptographic primitives and a non-committing and balanced KVS.

Overview. The scheme $\mathsf{EKVS} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ is described in detail in Figure 4.2 and, at a high level, works as follows. It makes black-box use of a key-value store $\mathsf{KVS} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$, a pseudo-random function F and a symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

The Gen algorithm takes as input a security parameter 1^k and uses it to generate a key K_1 for the pseudo-random function F and a key K_2 for the symmetric encryption scheme SKE. It then outputs a key $K = (K_1, K_2)$. The $\mathsf{Overlay}$, Alloc, $\mathsf{FrontEnd}$ and Daemon algorithms respectively execute $\mathsf{KVS.Overlay}$, $\mathsf{KVS.Alloc}$, $\mathsf{KVS.FrontEnd}$ and $\mathsf{KVS.Daemon}$ to generate and output the paramters ω, ψ and ϕ. The Put algorithm takes as input the secret key K and a label/value pair (ℓ, v). It first computes $t := F_{K_1}(\ell)$ and $e \leftarrow \mathsf{Enc}(K_2, v)$ and then executes $\mathsf{KVS.Put}(t, e)$. The Get algorithm takes as input the secret key K and a label ℓ. It computes $t := F_{K_1}(\ell)$ and executes $e \leftarrow \mathsf{KVS.Get}(t)$. It then outputs $\mathsf{SKE.Dec}(K, e)$.

Security. We now describe the leakage of EKVS. Intuitively, it reveals to the adversary the times at which a label is stored or retrieved with some probability. More formally, it is defined with the following *stateful* leakage function

- $\mathcal{L}_\varepsilon(\mathsf{DX}, (\mathsf{op}, \ell, v))$:
  1. if ℓ has never been seen
     (a) sample and store $b_\ell \leftarrow \mathsf{Ber}(\varepsilon)$
  2. if $b_\ell = 1$
     (a) if $\mathsf{op} = \mathtt{put}$ output $(\mathtt{put}, \mathsf{opeq}(\ell))$
     (b) else if $\mathsf{op} = \mathtt{get}$ output $(\mathtt{get}, \mathsf{opeq}(\ell))$
  3. else if $b_\ell = 0$
     (a) output $\perp$

where opeq is the *operation equality pattern* which reveals if and when a label was queried or put in the past.

We now state our main security Theorem and skip its proof because it is exactly the same as the proof of Theorem 3.4.2.

Theorem 4.5.1. *If $|I| \leq \theta$ and if KVS is RYW consistent, $(\varepsilon, \delta, \theta)$-balanced, has non-committing allocations and has label-independent allocation and front-end distributions, then EKVS is $\mathcal{L}_\varepsilon$-secure with probability at least $1 - \delta - \mathsf{negl}(k)$.*

Efficiency. The standard scheme does not add any overhead to time, round, communication and storage complexities of the underlying KVS.

Let $\mathsf{KVS} = (\mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ be a key-value store, $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key encryption scheme and F be a pseudo-random function. Consider the encrypted key-value store $\mathsf{EKVS} = (\mathsf{Gen}, \mathsf{Overlay}, \mathsf{Alloc}, \mathsf{FrontEnd}, \mathsf{Daemon}, \mathsf{Put}, \mathsf{Get})$ that works as follows:

- $\mathsf{Gen}(1^k)$:
  1. sample $K_1 \xleftarrow{\$} \{0,1\}^k$ and compute $K_2 \leftarrow \mathsf{SKE.Gen}(1^k)$
  2. output $K = (K_1, K_2)$
- $\mathsf{Overlay}(n)$:
  1. compute and output $\omega \leftarrow \mathsf{KVS.Overlay}(n)$
- $\mathsf{Alloc}(n, \omega, \rho)$:
  1. compute and output $\psi \leftarrow \mathsf{KVS.Alloc}(n, \omega, \rho)$
- $\mathsf{FrontEnd}(n, \omega)$:
  1. compute and output $\phi \leftarrow \mathsf{KVS.FrontEnd}(n, \omega)$
- $\mathsf{Daemon}(\omega, \psi, \rho, n)$:
  1. Execute $\mathsf{KVS.Daemon}(\omega, \psi, \rho, n)$
- $\mathsf{Put}(K, \ell, v)$:
  1. Parse K as (K_1, K_2)
  2. compute $t := F_{K_1}(\ell)$
  3. compute $e \leftarrow \mathsf{SKE.Enc}(K_2, v)$
  4. execute $\mathsf{KVS.Put}(t, e)$
- $\mathsf{Get}(K, \ell)$:
  1. Parse K as (K_1, K_2)
  2. Initialise $v := \perp$
  3. compute $t := F_{K_1}(\ell)$
  4. execute $e \leftarrow \mathsf{KVS.Get}(t)$
  5. if $e \neq \perp$, compute and output $v \leftarrow \mathsf{SKE.Dec}(K_2, e)$

Figure 4.2: The Standard EKVS Scheme

4.6 A Concrete Instantiation Based on Consistent Hashing

In this section, we analyze the security of the standard EKVS when its underlying KVS is instantiated with a consistent hashing based KVS (CH-KVS). We first give a brief overview of consistent hashing and then show that: (1) it has non-committing allocations in the random oracle model; and (2) it is balanced under two commonly used routing protocols.

Since Chord, a DHT we analyzed in the Chapter 3, is also based on consistent hashing, we will use a lot of machinery we developed for it to analyze CH-KVSs.

Setting up a CH-KVS. The setup of a CH-KVS is similar to that of Chord. A CH-KVS also arranges all 2^m addresses in its address space $\mathbf{A}$ in a ring and assigns to each active node $N \in \mathbf{C}$ an address $H_1(N)$ on the ring. The only difference is the way it computes the replicas for a label– like

Chord, it maps every label ℓ to $H_2(\ell)$ but unlike Chord, instead of storing ℓ at a single successor node of $H_2(\ell)$, it stores ℓ at ρ successor nodes of $H_2(\ell)$.

Formally, $\mathsf{addr}_\omega = H_1$ and $\mathsf{fe}_\phi = H_3$ as before, however, $\mathsf{replicas}_{\omega,\psi} = (\mathsf{succ}_{\chi_\mathbf{C}} \circ H_2, \ldots, \mathsf{succ}^\rho_{\chi_\mathbf{C}} \circ H_2)$. Recall that $\chi_\mathbf{C}$, called a *configuration*, contains a sequence $\{H_1(N_1), \ldots, H_1(N_n)\}$ of addressees assigned to active nodes in $\mathbf{C}$, and $\mathsf{succ}_{\chi_\mathbf{C}}$ is the *successor* function that assigns each address in $\mathbf{A}$ to its least upper bound in $\chi_\mathbf{C}$.

Routing protocols. There are two common routing protocols with CH-KVSs; each with trade-offs in storage and efficiency.

- *Multi-hop routing.* Based on H_1, the Daemon algorithm constructs a routing table by storing the addresses of the node's 2^ith successors where $0 \leq i \leq \log n$ (we refer the reader to [110] for more details). Note that a routing table contains at most $\log n$ other nodes. The routing protocol is fairly simple: given a message destined to a node N_d, a node N checks if $N = N_d$. If not, the node forwards the message to the node N' in its routing table with an address closest to N_d. Note that the route_ω map is deterministic given a fixed set of active nodes and it guarantees that any two nodes have a path of length at most $\log n$.

- *Zero-hop routing.* Based on H_1, the Daemon algorithm constructs a routing table by storing the addresses of all the other nodes in the routing table. Routing is then straightforward: given a message for N_d, simply forward it to the address of N_d. In short, for any two addresses s and d, $\mathsf{route}_\omega(s, d) = \{s, d\}$.

Storing and retrieving. When a client wants to execute a Get/Put operation on a label ℓ, it forwards the operation to the front-end node of ℓ. The front-end node executes the operation on the client's behalf as follows. It computes $\mathsf{replicas}(\ell)$ and forwards the operation to one of them. This replica is called the coordinator node. The coordinator then sends the operation to all (or a subset) the other replicas which then either update their state (on Put) or return a response back to the coordinator (on Get). In case more than one value is returned to the coordinator, it decides which value(s) is to be returned to the front-end. The choice of the coordinator node for a label ℓ varies from KVS to KVS. It can be a fixed node or a different node between requests for label ℓ. Either way, it is always a node chosen from the set of replicas. This guarantees that the visibility of a label (and hence the leakage) does not change between requests. KVSs also employ different synchronization mechanisms, like Merkle trees and read repairs to synchronize divergent replicas.

Non-committing allocation. Given a label ℓ and an address a, the allocation $(H_2, \mathbf{K})$ can be changed by programming the random oracle H_2 to output a when it is queried on ℓ.

Allocation distribution. We now describe the allocation distribution of CH-KVSs. Since CH-KVSs assign labels to addresses using a random oracle H_2, it follows that for all overlays $(H_1, \mathbf{C})$,

all labels $\ell \in \mathbf{L}$ and addresses $a \in \mathbf{A}$,

$$f_{H_2}(a) = \Pr[\, H_2(\ell) = a \,] = \frac{1}{|\mathbf{A}|},$$

which implies that CH-KVSs have label-independent allocations. From this it also follows that $\Delta(S)$ has a probability mass function

$$f_{\Delta(S)}(a) = \frac{f_\psi(a)}{\sum_{a \in S} f_\psi(a)} = \frac{1}{|\mathbf{A}|}\left(\frac{|S|}{|\mathbf{A}|}\right)^{-1} = \frac{1}{|S|}.$$

Before describing the visibility of nodes in CH-KVSs and analyzing their balance under zero-hop and multi-hop routing protocols, we define notation that will be useful in our analysis.

Notation. Recall that the *arc* of a node N is the set of addresses in $\mathbf{A}$ between N's predecessor and itself. More formally, we write $\mathsf{arc}_\chi(N) = (\mathsf{pred}_\chi(H_1(N)), \ldots, H_1(N)]$, where $\mathsf{pred}_\chi(N)$ is the *predecessor* function which assigns each address in $\mathbf{A}$ to its largest lower bound in χ. We extend the notion of arc of a node to ρ-arcs of a node. A ρ-arc of a node N is the set of addresses between N's ρth predecessor and itself. More formally, we write $\mathsf{arc}^\rho_\chi(N) = (\mathsf{pred}^\rho_\chi(H_1(N)), \ldots, H_1(N)]$, where $\mathsf{pred}^\rho_\chi(H_1(N))$ represents the predecessor function applied ρ times on $H_1(N)$. Intuitively, if H_2 hashes a label ℓ anywhere in ρ-arc of N, then N becomes one of the ρ replicas of ℓ. We denote by $\mathsf{maxareas}(\chi, x)$, the sum of the lengths (sizes) of x largest arcs in configuration χ. The *maximum area* of a configuration χ is equal to $\mathsf{maxareas}(\chi, \rho\theta)$. As we will later see, the maximum area is central to analyzing the balance of CH-KVSs.

4.6.1 Zero-hop CH-KVSs

In this section, we analyse the visibility and balance of zero-hop CH-KVSs.

Visible addresses. Given a fixed overlay $(H_1, \mathbf{C})$, an address $s \in \mathbf{A}$ and a node $N \in \mathbf{C}$, if the starting address is $s = H_1(N)$, then $\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) = \mathbf{A}$. This is because $H_1(N)$ lies on $\mathsf{route}_{\chi_\mathbf{C}}(s, a)$ for all $a \in \mathbf{A}$. Now for an address $s \in \mathbf{A}$ such that $s \neq H_1(N)$, we have

$$\begin{aligned}
\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) &= \left\{ \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N') : H_1(N) \in \mathsf{route}_{\chi_\mathbf{C}}(s, H_1(N')) \right\} \bigcup \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N) \\
&= \left\{ \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N') : H_1(N) \in \{s, H_1(N')\} \right\} \bigcup \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N) \\
&= \left\{ \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N') : H_1(N) = H_1(N') \right\} \bigcup \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N) \\
&= \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N)
\end{aligned}$$

where the second equality follows from the fact that $\mathsf{route}_{\chi_\mathbf{C}}(s, H_1(N')) = \{s, H_1(N')\}$, the third follows from the assumption that $H_1(N) \neq s$, and the fourth from the fact that $\mathsf{arc}^\rho_{\chi_\mathbf{C}}(N) = \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N')$ if $H_1(N) = H_1(N')$. Finally, for any set $S \subseteq \mathbf{C}$, $\mathsf{Vis}_{\omega,\mathbf{C}}(s, S) = \cup_{N \in S} \mathsf{Vis}_{\omega,\mathbf{C}}(s, N)$.

Balance of zero-hop CH-KVSs. We are now ready to analyze the balance of zero-hop CH-KVSs.

Theorem 4.6.1. *Let* $\mathbf{C} \subseteq \mathbf{N}$ *be a set of active nodes. If* $\mathsf{maxareas}(\chi_{\mathbf{C}}, \rho\theta) \leq \lambda$*, then* $\chi_{\mathbf{C}}$ *is* (ε, θ)*-balanced with*

$$\varepsilon = \frac{\theta}{|\mathbf{C}|} + \frac{\lambda}{|\mathbf{A}|}$$

Proof. Let $n = |\mathbf{C}|$ and S be a set of addresses of at most θ nodes in $\mathbf{C}$. For all $\ell \in \mathbf{L}$, we define $\mathcal{E}_1$ as the event that address of at least one of the replicas of ℓ is in S and $\mathcal{E}_2$ as the event that at least one of the addresses in S is on the path to a replica of ℓ. Precisely,

$$\mathcal{E}_1 = \{S \cap \mathsf{replicas}(\ell) \neq \emptyset\}, \qquad \mathcal{E}_2 = \{S \cap \mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell)) \neq \emptyset\}$$

where for visual clarity, we represent $\cup_{N \in \mathsf{replicas}(\ell)} \mathsf{route}(\mathsf{fe}(\ell), N)$ by $\mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell))$. For CH-KVS, we then have that,

$$\Pr\left[\, \mathsf{replicas}_{\chi_{\mathbf{C}}}(\ell) \cap \mathsf{Vis}_{\chi_{\mathbf{C}}}(\mathsf{fe}(\ell), S) \neq \emptyset \,\right] = \Pr\left[\, \mathcal{E}_1 \vee \mathcal{E}_2 \,\right] = \Pr\left[\, \mathcal{E}_2 \,\right], \tag{4.1}$$

where the second equality is because $\mathcal{E}_1$ is contained in $\mathcal{E}_2$. This is true because the last node on the paths to the replicas is a replica itself. We now look at event $\mathcal{E}_2$.

$$\begin{aligned}
\Pr\left[\, \mathcal{E}_2 \,\right] &\leq \Pr\left[\, S \cap \{\mathsf{fe}(\ell), \mathsf{replicas}(\ell)\} \neq \emptyset \,\right] \\
&= \Pr\left[\, \mathsf{fe}(\ell) \in S \bigvee S \cap \mathsf{replicas}(\ell) \neq \emptyset \,\right] \\
&\leq \Pr\left[\, \mathsf{fe}(\ell) \in S \,\right] + \Pr\left[\, S \cap \mathsf{replicas}(\ell) \neq \emptyset \,\right] \qquad (4.2)
\end{aligned}$$

where the first inequality follows from the fact that, when using zero-hop routing, $\mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell))$ $= \{\mathsf{fe}(\ell), \mathsf{replicas}(\ell)\}$. We now compute the probability that one of the addresses in S is the front-end of ℓ.

$$\Pr\left[\, \mathsf{fe}(\ell) \in S \,\right] \leq \sum_{a \in S} \Pr\left[\, \mathsf{fe}(\ell) = a \,\right] = \sum_{a \in S} \Pr\left[\, H_3(\ell) = a \,\right] = \sum_{a \in S} \frac{1}{n} \leq \frac{\theta}{n}, \tag{4.3}$$

where the first inequality follows from the union bound and the last inequality follows from the fact that H_3 is a uniform random function over the addresses of active nodes. We now compute the probability that an address in S is the address of a replica of ℓ.

$$\begin{aligned}
\Pr\left[\, S \cap \mathsf{replicas}(\ell) \neq \emptyset \,\right] &= \Pr\left[\, H_2(\ell) \in \cup_{N : H_1(N) \in S} \mathsf{arc}^{\rho}_{\chi_{\mathbf{C}}}(N) \,\right] \\
&= \frac{|\cup_{N : H_1(N) \in S} \mathsf{arc}^{\rho}_{\chi_{\mathbf{C}}}(N)|}{|\mathbf{A}|} \\
&\leq \frac{\lambda}{|\mathbf{A}|}, \qquad (4.4)
\end{aligned}$$

where the first equality follows from the fact that an address in S is the address of one of the ρ replicas of ℓ only if H_2 maps ℓ in its ρ-arc, and the inequality follows from the assumption of the theorem. Finally, the Theorem follows by plugging Eqs. 4.3 and 4.4 into Eqn. 4.2.

■

Corollary 4.6.2. *Let* $\mathbf{C}$ *be a set of active nodes. For all* $\rho\theta \leq |\mathbf{C}|/e$*, a zero-hop CH-KVS is* $(\varepsilon, \delta, \theta)$*-balanced for*

$$\varepsilon = \frac{\theta}{|\mathbf{C}|}\left(1 + 6\rho \log\left(\frac{|\mathbf{C}|}{\rho\theta}\right)\right) \quad \text{and} \quad \delta = \frac{1}{|\mathbf{C}|^2} + (e^{-\sqrt{|\mathbf{C}|}} \cdot \log|\mathbf{C}|)$$

Proof. Recall that Lemma 3.5.4 upper bounds the sum of the lengths of the x largest arcs in a configuration χ in Chord. The sum is denoted by $\mathsf{maxareas}(\chi, x)$. Since Chord is also based on consistent hashing, we use the lemma to bound the maximum area of CH-KVSs by substituting $x = \rho\theta$. Substituting $x = \rho\theta$ in Corollary 3.5.4, we know that for $\rho\theta \leq |\mathbf{C}|/e$,

$$\Pr[\,\mathsf{maxareas}(\chi_{\mathbf{C}}, \rho\theta) \leq \lambda\,] \geq 1 - \delta \quad \text{for} \quad \lambda = \frac{6|\mathbf{A}|\rho\theta}{|\mathbf{C}|}\log\frac{|\mathbf{C}|}{\rho\theta}$$

and δ as stated in the Theorem statement. Therefore, by Theorem 4.6.1, we conclude that for $\rho\theta \leq |\mathbf{C}|/e$,

$$\Pr[\,(H_1, \mathbf{C}) \text{ is } (\varepsilon, \theta)\text{-balanced}\,] \geq 1 - \delta \quad \text{for} \quad \varepsilon = \frac{\theta}{|\mathbf{C}|} + \frac{\lambda}{|\mathbf{A}|}.$$

The Corollary follows by substituting the value of λ in the last equation.

■

Remark. It follows from Corollary 4.6.2 that

$$\varepsilon = O\left(\frac{\rho\theta}{|\mathbf{C}|}\log\left(\frac{|\mathbf{C}|}{\rho\theta}\right)\right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. Note that assigning labels uniformly at random to ρ nodes would achieve $\varepsilon = \rho\theta/|\mathbf{C}|$ so zero-hop CH-KVSs balance data fairly well.

The Security of a Zero-Hop CH-KVS based EKVS. In the following Corollary, we formally state the security of the standard scheme when its underlying KVS is instantiated with a zero-hop CH-KVS.

Corollary 4.6.3. *If* $|\mathbf{L}| = \Theta(2^k)$*,* $|I| \leq |\mathbf{C}|/(\rho e)$*, and if* EKVS *is instantiated with a RYW zero-hop CH-KVS, then it is* $\mathcal{L}_\varepsilon$*-secure with probability at least* $1 - 1/|\mathbf{C}|^2 - (e^{-\sqrt{|\mathbf{C}|}} \cdot \log|\mathbf{C}|) - \mathsf{negl}(k)$ *in the random oracle model, where*

$$\varepsilon = \frac{|I|}{|\mathbf{C}|}\left(1 + 6\rho\log\left(\frac{|\mathbf{C}|}{\rho|I|}\right)\right).$$

The corollary follows from Theorem 4.5.1, Corollary 4.6.2 and the fact that CH-KVS has non-committing allocations when H_2 is modeled as a random oracle.
From the discussion of Corollary 4.6.2, we know,

$$\varepsilon = O\left(\frac{\rho|I|}{|\mathbf{C}|}\log\left(\frac{|\mathbf{C}|}{\rho|I|}\right)\right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. Setting $|I| = |\mathbf{C}|/(\rho\alpha)$, for some $\alpha \geq e$, we have $\varepsilon = O(\log(\alpha)/\alpha)$. Recall that, on each query, the leakage function leaks the operation equality with probability at most

ε. So intuitively this means that the adversary can expect to learn the operation equality of an $O(\log(\alpha)/\alpha)$ fraction of client operations if $\rho|I| = |\mathbf{C}|/\alpha$. Note that this confirms the intuition that distributing data suppresses its leakage.

4.6.2 Multi-hop CH-KVSs

In this section, we analyse the visibility and balance of multi-hop CH-KVSs. Since most of the details are similar to what was in the last section, we keep the description high level.

Visible addresses. Given a fixed overlay $(H_1, \mathbf{C})$, an address $s \in \mathbf{A}$ and a node $N \in \mathbf{C}$, if the starting address is $s = H_1(N)$, then $\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) = \mathbf{A}$. For an address $s \in \mathbf{A}$ such that $s \neq H_1(N)$, we have

$$\mathsf{Vis}_{\chi_\mathbf{C}}(s, N) = \left\{ \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N') : H_1(N) \in \mathsf{route}_{\chi_\mathbf{C}}(s, H_1(N')) \right\} \bigcup \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N)$$

Finally, for any set $S \subseteq \mathbf{C}$, $\mathsf{Vis}_{\omega,\mathbf{C}}(s, S) = \cup_{N \in S} \mathsf{Vis}_{\omega,\mathbf{C}}(s, N)$.

Balance of multi-hop CH-KVSs. We now analyze the balance of multi-hop CH-KVSs.

Theorem 4.6.4. *Let $\mathbf{C} \subseteq \mathbf{N}$ be a set of active nodes. If $\mathsf{maxareas}(\chi_\mathbf{C}, \rho\theta) \leq \lambda$, then $\chi_\mathbf{C}$ is (ε, θ)-balanced with*

$$\varepsilon = \frac{\rho\theta \log |\mathbf{C}|}{|\mathbf{C}|} + \frac{\lambda}{|\mathbf{A}|}$$

Proof. Let $n = |\mathbf{C}|$ and let S be a set of addresses of at most θ nodes in $\mathbf{C}$. For all $\ell \in \mathbf{L}$, we define $\mathcal{E}_1$ as the event that the address of at least one of the replicas of ℓ is in S and $\mathcal{E}_2$ as the event that at least one of the addresses in S is on the path to a replica of ℓ. Precisely,

$$\mathcal{E}_1 = \{S \cap \mathsf{replicas}(\ell) \neq \emptyset\}, \qquad \mathcal{E}_2 = \{S \cap \mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell)) \neq \emptyset\}$$

where for visual clarity, we represent $\cup_{N \in \mathsf{replicas}(\ell)} \mathsf{route}(\mathsf{fe}(\ell), N)$ by $\mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell))$. For CH-KVS, we then have that,

$$\Pr\left[\, \mathsf{replicas}_{\chi_\mathbf{C}}(\ell) \cap \mathsf{Vis}_{\chi_\mathbf{C}}(\mathsf{fe}(\ell), S) \neq \emptyset \,\right] = \Pr\left[\, \mathcal{E}_1 \vee \mathcal{E}_2 \,\right] \leq \Pr\left[\, \mathcal{E}_1 \,\right] + \Pr\left[\, \mathcal{E}_2 \,\right], \tag{4.5}$$

We first bound the probability of $\mathcal{E}_1$,

$$\Pr\left[\, \mathcal{E}_1 \,\right] = \Pr\left[\, S \cap \mathsf{replicas}(\ell) \neq \emptyset \,\right] = \Pr\left[\, H_2(\ell) \in \cup_{N : H_1(N) \in S} \mathsf{arc}^\rho_{\chi_\mathbf{C}}(N) \,\right] \leq \frac{\lambda}{|\mathbf{A}|} \tag{4.6}$$

This is the same as Equation 4.4 of Theorem 4.6.1. We now bound $\mathcal{E}_2$. By the union bound and the law of total probability, we have that,

$$\begin{aligned}
\Pr\left[\, \mathcal{E}_2 \,\right] &= \Pr\left[\, S \cap \mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell)) \neq \emptyset \,\right] \\
&\leq \sum_{N \in S} \Pr\left[\, N \in \mathsf{route}(\mathsf{fe}(\ell), \mathsf{replicas}(\ell)) \,\right] \\
&= \sum_{N \in S} \sum_{R \in \mathbf{C}^\rho} \Pr\left[\, N \in \mathsf{route}(\mathsf{fe}(\ell), R) \mid \mathsf{replicas}(\ell) = R \,\right] \cdot \Pr\left[\, \mathsf{replicas}(\ell) = R \,\right]
\end{aligned} \tag{4.7}$$

But note that,

$$\Pr[\, N \in \mathsf{route}(\mathsf{fe}(\ell), R) \mid \mathsf{replicas}(\ell) = R \,] \leq \sum_{r \in R} \Pr[\, N \in \mathsf{route}(\mathsf{fe}(\ell), r) \,] \leq \sum_{r \in R} \frac{\log n}{n} = \frac{\rho \log n}{n}$$

where the last inequality follows from the fact that path lengths in multi-hop CH-KVS are at most $\log n$. Substituting this in Eq. (4.7) we get,

$$\begin{aligned}
\Pr[\, \mathcal{E}_2 \,] &\leq \sum_{N \in S} \sum_{R \in \mathbf{C}^\rho} \frac{\rho \log n}{n} \cdot \Pr[\, \mathsf{replicas}(\ell) = R \,] \\
&= \sum_{N \in S} \frac{\rho \log n}{n} \sum_{N' \in \mathbf{C}^\rho} \Pr[\, \mathsf{replicas}(\ell) = R \,] \\
&= \sum_{N \in S} \frac{\rho \log n}{n} \\
&= \frac{\rho\theta \log n}{n} \qquad (4.8)
\end{aligned}$$

Finally, the Theorem follows by plugging Eqs. (4.6) and (4.8) into Eq. (4.5).

■

Corollary 4.6.5. *Let* $\mathbf{C}$ *be a set of active nodes. For all* $\rho\theta \leq |\mathbf{C}|/(e \log |\mathbf{C}|)$*, a multi-hop CH-KVS is* $(\varepsilon, \delta, \theta)$*-balanced for*

$$\varepsilon = \frac{\rho\theta}{|\mathbf{C}|}\left(\log |\mathbf{C}| + 6 \log\left(\frac{|\mathbf{C}|}{\rho\theta}\right)\right) \quad \text{and} \quad \delta = \frac{1}{|\mathbf{C}|^2} + (e^{-\sqrt{|\mathbf{C}|}} \cdot \log |\mathbf{C}|)$$

The Corollary follows directly from Corollary 3.5.4 and Theorem 4.6.4.
Notice that multi-hop CH-KVSs are not only less balanced than zero-hop CH-KVSs but also tolerate a lesser number of corruptions. This is the case because in a multi-hop CH-KVS there is a higher chance that an adversary sees a label since the routes are larger.

Remark. It follows from Corollary 4.6.5 that

$$\varepsilon = O\left(\frac{\rho\theta}{|\mathbf{C}|} \log |\mathbf{C}|\right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. As discussed earlier, the optimal balance is $\varepsilon = \rho\theta/|\mathbf{C}|$, which is achieved when labels are assigned uniformly at random to ρ nodes. Note that balance of multi-hop CH-KVSs is only $\log |\mathbf{C}|$ factor away from optimal balance which is very good given that the optimal balance is achieved with no routing at all.

The Security of a Multi-Hop CH-KVS based EKVS. In the following Corollary, we formally state the security of the standard scheme when its underlying KVS is instantiated with a multi-hop CH-KVS.

Functionality $F^{\mathcal{L}}_{\mathsf{mKVS}}$

$F^{\mathcal{L}}_{\mathsf{mKVS}}$ stores a dictionary DX initialized to empty, a time counter τ initialized to 0 and proceeds as follows, running with c clients $\mathcal{C}_1, \ldots, \mathcal{C}_c$, n nodes $N_1, \ldots, N_n$ and a simulator Sim:

- $\mathsf{Put}(\ell, v)$: Upon receiving a label/value pair (ℓ, v) from some client $\mathcal{C}_i$, it increments τ by 1, sets $\mathsf{DX}[\ell \,||\, \tau] := v$, and sends the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{put}, \ell, v))$ to the simulator Sim.
- $\mathsf{Get}(\ell)$: Upon receiving a label ℓ from the client $\mathcal{C}_i$, it increments τ by 1, sends the leakage $\mathcal{L}(\mathsf{DX}, (\mathtt{get}, \ell, \perp))$ to the simulator Sim, and when it receives a message τ' from the simulator, it returns $\mathsf{DX}[\ell \,||\, \tau']$ to $\mathcal{C}_i$.

Figure 4.3: $F^{\mathcal{L}}_{\mathsf{mKVS}}$: The ideal multi-user KVS functionality parameterized with leakage function $\mathcal{L}$.

Corollary 4.6.6. *If $|\mathbf{L}| = \Theta(2^k)$, $|I| \leq |\mathbf{C}|/(\rho e \log |\mathbf{C}|)$, and if EKVS is instantiated with a RYW multi-hop CH-KVS, then it is $\mathcal{L}_\varepsilon$-secure with probability at least $1 - 1/|\mathbf{C}|^2 - (e^{-\sqrt{|\mathbf{C}|}} \cdot \log |\mathbf{C}|) - \mathsf{negl}(k)$ in the random oracle model, where*

$$\varepsilon = \frac{\rho |I|}{|\mathbf{C}|} \left(\log |\mathbf{C}| + 6 \log \left(\frac{|\mathbf{C}|}{\rho |I|} \right) \right).$$

From the discussion of Theorem 4.6.5, we know that,

$$\varepsilon = O\left(\frac{\rho |I|}{|\mathbf{C}|} \log |\mathbf{C}| \right)$$

and $\delta = O(1/|\mathbf{C}|^2)$. Setting $|I| = |\mathbf{C}|/(\rho \alpha \log |\mathbf{C}|)$, for some $\alpha \geq e$, we have $\varepsilon = O(1/\alpha)$, which intuitively means that the adversary can expect to learn the operation equality of an $O(1/\alpha)$ fraction of client operations.

4.7 The Standard EKVS Scheme in the Multi-User Setting

We now analyze the security of the standard scheme in a more general setting, i.e., where we no longer require the underlying KVS to satisfy RYW and where we no longer assume that a single client operates on the data. We call this setting the *multi-user* setting where multiple clients operate on the same data concurrently. We start by extending our security definition to the multi-user setting and then analyze the security of the standard scheme (from Figure 4.2) in this new setting.

The ideal multi-user KVS functionality. The ideal multi-user KVS functionality $F^{\mathcal{L}}_{\mathsf{mKVS}}$ is described in Figure 4.3. The functionality stores all the values that were ever written to a label. It also associates a time τ with every value indicating when the value was written. On a Get operation, it sends leakage to the simulator which returns a time τ'. The functionality then returns the value associated with τ' to the client. Notice that, unlike single-user ideal functionality $\mathcal{F}^{\mathcal{L}}_{\mathsf{KVS}}$, the multi-user ideal functionality can be influenced by the simulator.

Security definition. The real and ideal experiments are the same as in Section 3.4.1 with the following differences. First, the experiments are executed not with a single client but with c clients

$\mathcal{C}_1 \dots \mathcal{C}_c$; second, the environment adaptively sends operations to all these clients; and third, the ideal functionality of Figure 3.1 is replaced with the ideal functionality described in Figure 4.3.

4.7.1 Security of the Standard Scheme

We now analyze the security of the standard scheme when its underlying KVS is instantiated with a KVS that does not necessarily satisfy RYW consistency. We start by describing its stateful leakage function.

- $\mathcal{L}(\mathsf{DX}, (\mathsf{op}, \ell, v))$:
  1. if $\mathsf{op} = \mathtt{put}$ output $(\mathtt{put}, \mathsf{opeq}(\ell))$
  2. else if $\mathsf{op} = \mathtt{get}$ output $(\mathtt{get}, \mathsf{opeq}(\ell))$

where opeq is the operation equality pattern which reveals if and when a label was queried or put in the past.

Single-user vs. multi-user leakage. Notice that the leakage profile achieved in the multi-user setting is a function of all the labels whereas the leakage profile achieved in the single-user setting was only a function of the labels that were (exclusively) stored and routed by the corrupted nodes. In particular, this implies that the multi-user leakage is worse than the single-user leakage and equivalent to the leakage achieved by standard (non-distributed) schemes. In following, we will refer to the labels stored and routed exclusively by honest nodes as "honest labels" and to all the other labels as "corrupted labels".

The reason that the single-user leakage is independent of the honest labels is because of the RYW consistency of the underlying KVS. More precisely, RYW consistency guarantees that for a given label, the user will read the latest value that it stored. This implies that the value it reads will be independent of any other label, including the corrupted labels. This is not the case, however, in the multi-user setting where RYW consistency does not guarantee that the honest labels will be independent of the corrupted labels. To see why, consider the following example. Let ℓ_1 be a corrupted label and let ℓ_2 be an honest label. Assume that both ℓ_1 and ℓ_2 initially have the value 0. Now consider the two sequences of operations executed by clients C_1 and C_2 shown in Figure 4.4. Notice that both sequences are RYW consistent (this is the case because they satisfy a stronger consistency guarantee called *sequential consistency*). However, in sequence 1, $\mathsf{Get}(\ell_2)$ can output both 0 or 1 whereas, in sequence 2, if $\mathsf{Get}(\ell_1)$ outputs a 0, then $\mathsf{Get}(\ell_2)$ can only output 1. This example points out that operations on corrupted labels can impact operations on honest labels. Capturing exactly how operations on one label can effect operations on other labels for different consistency guarantees is challenging but might be helpful in designing solutions with better leakage profiles. We leave this as an open problem. Alternatively, it would be interesting to know if there is some consistency notion one could assume (in the multi-user setting) under which a better leakage profile could be achieved.

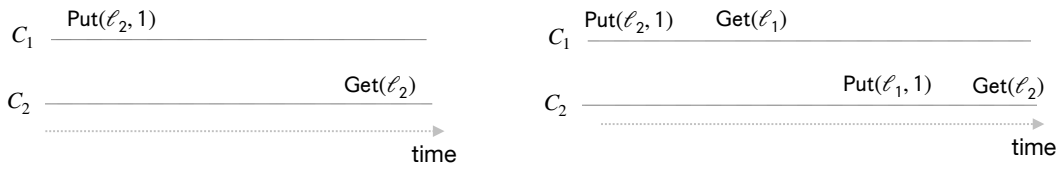


Figure 4.4: Sequence 1 is on the left and Sequence 2 is on the right.

Security. We now state our security theorem.

Theorem 4.7.1. EKVS *is* $\mathcal{L}$*-secure with probability at least* $1 - \mathsf{negl}(k)$.

4.8 Conclusions and Future Work

In this work, we study end-to-end encryption in the context of KVSs. We formalize the security properties of the standard scheme in both the single-user and multi-user settings. We then use our framework to analyze the security of the standard scheme when its underlying KVS is instantiated with consistent hashing based KVS (with zero-hop and multi-hop routing). We see our work as an important step towards designing provably-secure end-to-end encrypted distributed systems like off-chain networks, distributed storage systems, distributed databases and distributed caches.

Our work motivates several open problems and directions for future work.

Relationship between consistency guarantees and leakage. Recall that the standard scheme leaks the operation equality of all the labels in the multi-user setting (with no assumption on the consistency guarantees). However, if the underlying KVS satisfies RYW consistency, the scheme only leaks the operation equality of a subset of labels but in a single-user setting. The most immediate question is whether the leakage can be improved in the multi-user setting by assuming a stronger consistency guarantee.

We however believe that even assuming linearizability, which is much stronger than RYW consistency, the standard scheme would still leak more in the multi-user setting than what it would in the single-user setting with RYW consistency. The question then is to find a lower bound on leakage in the multi-user setting.

Beyond CH-KVS. Another direction is to study the security of the standard EKVS when it is instantiated with a KVS that is not based on consistent hashing or on the two routing schemes that we described. Instantiations based on Kademlia [88] and Koorde [70] would be particularly interesting due to the former's popularity in practice and the latter's theoretical efficiency. Because Koorde uses consistent hashing in its structure (though its routing is different and based on De Bruijn graphs) the bounds we introduce in this work to study CH-KVS's balance might find use in

analyzing Koorde. Kademlia, on the other hand, has a very different structure than CH-KVSs so it is likely that new custom techniques and bounds are needed to analyze its balance.

New EKVS constructions. A third direction is to design new EKVS schemes with better leakage profiles. Here, a "better" profile could be the same profile $\mathcal{L}_\varepsilon$ achieved in this work but with a smaller ε than what we show. Alternatively, it could be a completely different leakage profile. This might be done, for example, by using more sophisticated techniques from structured encryption and oblivious RAMs.

EKVSs in the transient setting. Another important direction of immediate practical interest is to study the security of EKVSs in the transient setting. As mentioned in Section 4.3, in transient setting, nodes are not known a-priori and can join and leave at any time. This setting is particularly suited to peer-to-peer networks and permissionless blockchains. Agarwal and Kamara [24] study DHTs in the transient setting and it would be intersting to extend their work to transient KVSs as well.

Stronger adversarial models. Our security definitions are in the standalone model and against an adversary that makes static corruptions. Extending our work to handle arbitrary compositions (e.g., using universal composability [46]) and adaptive corruptions would be very interesting.

Chapter 5

Encrypted Blockchain Databases

5.1 Introduction

Blockchains are decentralized tamper-proof append-only data stores. Since their introduction by Nakamoto in the context of crypto-currencies [93], blockchains have received a lot of attention from research and industry due to their potential use tamper-proof distributed storage platforms. An emerging and important class of blockchain technologies are *blockchain databases*. These are blockchain-like in the sense that they are decentralized and tamper-proof but database-like in the sense that they store complex data types, provide (relatively) low latency, and support complex queries. Blockchain databases are a crucial technology for the development of non-trivial smart contracts, distributed applications and marketplaces. Examples of commercial and research blockchain databases include Bigchain DB [89], Bluzelle [14] and [94].

As blockchain databases gain wider adoption, concerns over the confidentiality of the data they manage will increase. Already, several projects aim to use blockchains to store sensitive data like electronic healthcare and financial records, legal documents (e.g., wills) and customer data. But the decentralized nature of blockchains—where data is highly replicated and stored on untrusted nodes—makes it a particularly poor solution for storing sensitive data.

Encrypted blockchain DBs. In this work, we consider the problem of end-to-end *encrypted blockchain databases*. With such a system, a client can encrypt its database before storing it on the blockchain. To query it, the client uses its secret key and executes a query protocol with the blockchain. Encrypted blockchain DBs are a form of encrypted database as studied in the encrypted search literature. In theory they could be designed using various cryptographic primitives each of which achieve different tradeoffs between query efficiency, storage overhead, communication complexity, leakage and query expressiveness.

Encrypted multi-maps. A multi-map (MM) is a data structure that stores label/tuple pairs and supports get and put operations. Gets take as input a label and return the associated tuple and

puts take as input a label/tuple pair and stores it. Multi-maps are generalization of dictionaries which only store label/value pairs. MMs capture the functionalities of important data structures like inverted indices but can also be used to represent NoSQL databases like key-value stores (e.g., DynamoDB) and document stores (e.g., MongoDB). An encrypted multi-map (EMM) is an end-to-end encrypted multi-map data structure that supports puts and gets over encrypted data. EMMs have have received a lot of attention in the encrypted search literature because they enable sub-linear search on encrypted data [53], encrypted graph databases [52] and encrypted relational databases [72]. We note that an encrypted NoSQL blockchain database can be trivially constructed from a decentralized/blockchain EMM since both key-value stores and document databases can be represented as dictionaries. Because of this, in this work, we focus on the problem of designing of blockchain EMMs.

Custom vs. legacy-friendly designs. There are two approaches one could take to design a blockchain EMM. The first is to build a custom system from the ground up. The advantage of this approach is that the blockchain and the encrypted search techniques can be co-designed to optimize performance. Another approach is to design a solution that is legacy-friendly in the sense that it can be used on top of pre-existing blockchains. The advantage of this approach is that the resulting system can benefit from the underlying blockchain's network in terms of size and adoption. The disadvantage of this approach is that it introduces several technical challenges.

Challenges. The first challenge is simply to find a way to store the encrypted database on a blockchain. Most existing blockchains were designed to store financial transactions or the state of smart contracts but not databases and the data structures that support them. The second challenge is in achieving dynamism; that is, adding, deleting and editing data. One of the core properties of blockchains is that they are tamper-proof which makes database deletion operations particularly difficult. The third challenge is to achieve efficiency both with respect to queries and updates.

5.1.1 Our Contributions

In this work, we show how to design practical legacy-friendly encrypted blockchain databases. We make several contributions.

Append-only data stores. Our blockchain EMM constructions can work on any blockchain. To achieve this level of generality, we use a simple abstraction called an append-only data store (ADS) that captures the properties and functionality of blockchains that we need. At a high level, an ADS stores address/entry pairs but where the address is determined by the structure—as opposed to a dictionary where the label is chosen. ADSs are append-only so they only support get and put operations. By designing EMMs based on ADSs we ensure that our constructions can be implemented and used on any blockchain. An alternative approach would be to store an entire EMM as the state of a smart contract and to implement the query and update operations as a smart

contract. There are two limitations to this approach. First, it is not general-purpose since: (1) many blockchains do not support smart contracts; and (2) many smart contract platforms do not maintain state across transactions. The second limitation is that it is expensive since smart contract platforms require payment not only for storing data and code but also for executing code (and the more complex the code is, the higher the cost). Our approach, on the other hand, is general and lower cost since we can store ADS entries in transactions as opposed to smart contracts [1] and don't need to execute any code on the blockchain.

A list-based construction. Our first construction, LSX, stores every element of a multi-map in the ADS but super-imposes a virtual linked list for each tuple. Given a tuple $(v_1, \ldots, v_n)$ associated to a label ℓ, the values are first stored in blocks $(B^{(1)}, \ldots, B^{(m)})$. Block $B^{(i)}$ is then concatenated with the address of $B^{(i-1)}$, encrypted and stored in the ADS. The address of $B^{(m)}$ is then stored locally by the client. To query the EMM on a label ℓ, the client recovers the address of the tail block and queries the ADS for it. This results in the client learning $B^{(n)}$ and the address of $B^{(n-1)}$ which can now be queried; and so on and so forth. To achieve dynamism, the scheme uses lazy deletion: all the added and deleted values are marked as added or deleted and deletion is only performed at query time by removing the values marked as deleted from the output. This scheme has several shortcomings including query complexity that is linear in the number of deleted items and a put operation that requires a linear number of rounds (in the size of the tuple). The latter is particularly costly when the ADS is instantiated with a blockchain because the latency of a round is equivalent to the time it takes for a transaction to stabilize, which can be very high.

Like any encrypted search solution, our schemes achieve tradeoffs between efficiency and leakage. We formally analyze the security of our constructions and prove that they achieve standard leakage profiles. More precisely, LSX's query leakage reveals if and when the queried label has been edited in the past. Its add and delete leakages, on the other hand, reveal only the size of the tuple, which implies that LSX is forward-private [108, 39].

A tree-based construction. Our second construction, TRX, improves on LSX's round complexity for puts. It does this by super-imposing a binary tree instead of a list. Roughly speaking, given blocks $(B^{(1)}, \ldots, B^{(m)})$, each block is concatenated with the addresses of a left and a right block. These blocks can be any two blocks that have been previously stored but not linked to. For dynamism, the scheme also uses lazy deletes. The advantage of this scheme is that put operations now require only a logarithmic number of rounds since all the blocks at a given tree depth can be inserted into the ADS in parallel (since their children have already been inserted and their addresses are now known). TRX achieves the same leakage profile as LSX.

A patched construction. Our third construction, PAX, improves on the asymptotic query complexity of LSX. It does this by super-imposing additional (virtual) structures on the ADS. The first

[1] On Ethereum, it costs around $2,200$ gas to store a 32 bytes in a transaction whereas it costs around $20,000$ gas to store it in a smart contract.

are what we refer to as *patches*. These are address pairs that allow the query algorithm to skip values that are deleted. To guarantee correctness and to achieve optimal query complexity, these patches have to be used and managed very carefully. We achieve this by super-imposing a binary search tree on the patches themselves that allows us to find patches quickly and introduce a set of techniques to manage this patch tree.

PAX's leakage profile is slightly worse than LSX's and TRX's. Its query leakage reveals if and when an addition was made to the label and its add leakage reveals only the size of the tuple. In particular, this means PAX's adds are forward private. Its deletions, however, are not and they reveal if and when the label was added to, and if and when the tuple values were added in the past. Another limitation of PAX is that it does not support packing so, even though its asymptotic query complexity is optimal, its concrete efficiency is only moderate (as our experiments reveal). Nonetheless, we believe that PAX is an interesting construction due to the techniques it introduces and its asymptotic optimality. Furthermore, if it can be extended to handle packing it would be very efficient in practice.

Blockchain instantiations of ADSs. We show how to use the Ethereum and Algorand blockchains to instantiate an ADS. At a high-level, we store a value v by creating a transaction that stores v and sending it to the blockchain so that it gets mined into a block. In the context of blockchains, the address of a value can be instantiated in one of two ways: (1) as a transaction hash, which the client can compute *before* the block is mined; or (2) as the block number (along with a transaction hash) which the client can only use *after* the transaction has been mined and the block is stable. Using transaction hashes as addresses is more efficient but, in some cases, infeasible because some nodes might not support transaction lookups by hash. Because of this, we also study how choosing one instantiation over the other effects the efficiency of our schemes.

Empirical evaluation. We implemented our schemes on the Algorand testnet and evaluated them under a variety of different settings. We varied the sizes of the multi-maps and the querying and deletion patterns. We instantiated the ADS addresses using transaction hashes and block numbers. At a high level, we found that TRX performs better than LSX when addresses are instantiated with block numbers. However, if transaction hashes are used, we found no difference between the two. We also found that, as expected, for workloads with a lot of delete operations, PAX outperforms the other schemes. For other workloads, however, PAX performs worse than the other schemes due to its inability to pack multiple values in a single transaction. [2]

[2] We implemented our schemes for the Ethereum testnet as well but could not run any experiments due to what we believe is an DDoS or anti-spam mechanism. We contacted the Ethereum foundation about it but never heard back. We expect, however, to see similar trends as our results from the Algorand testnet.

5.2 Related Work

Blockchain databases. Recently, database and blockchain technologies have mutually influenced each other. Some databases have adopted blockchain features such as decentralization, tamper-resistance and auditability [95, 18, 15, 89, 19, 104], while some blockchains have adopted database features like low latency and expressive queries [67, 27, 59, 114, 112, 34, 81]. The latter, (i.e., blockchain DBs) work by either storing data in traditional database and using blockchains for book-keeping purposes [114, 112, 34, 81]; or by introducing an additional database layer on top of an existing blockchain [67, 27, 59].

Privacy in blockchains. Blockchains are being designed for a variety of uses and domains such as government, health and IoT [36, 30, 115, 103]. Since many blockchains are public and store sensitive data, privacy has always been a concern. This has led to the adoption of various cryptographic techniques like zero-knowledge proofs, [91, 105], secure multi-party computation [112, 79, 12], secret sharing [32, 12], encryption [86, 27, 80], commitment schemes [41, 11, 87] and access controls [27, 96, 97] to blockchains. We refer the readers to [102, 116] for a comprehensive survey.

Recently, Benhamouda et al. [35] considered the problem of storing and using a secret on blockchain. We note the goal of our work is different from theirs but complimentary. In this work, we are concerned with storing and querying a database of sensitive information whereas in [35] the goal is to store and use a secret like an encryption or a signing key. The two approaches could be combined as follows. Suppose we had two blockchains bc_1 and bc_2. Our blockchain EMMs could be used to store and manage a database on bc_1 while the techniques from [35] could be used to store the blockchain EMM's secret key on bc_2 and to execute the query, add and delete operations.

Verifiable SSE via blockchains. Blockchains have also been used in the context of encrypted search. Several works [44, 43, 68, 120] propose to use blockchains to desgin verifiable searchable symmetric encryption (VSSE) schemes. A VSSE scheme is a searchable encryption scheme where the client can verify the correctness of the query results. Traditional VSSE constructions rely on cryptographic primitives like message authentication codes (MAC) and digital signatures. These works replace the server by a smart contract and rely on the consensus mechanism of the latter to provide a guarantee of correctness.

5.3 Preliminaries

Append-only data stores. An append-only data store ADS is a special case of a dictionary data structure in which every inserted label/value pair cannot be removed without impacting the integrity of the entire structure. That is, the structure can insert new label/value pairs, but the existing ones are immutable and cannot be modified. We provide below a formal description of this data structure.

Definition 5.3.1 (Append-only data store)**.** *An append-only data store* $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ *consists of three algorithms that work as follows:*

- $\mathsf{ADS} \leftarrow \mathsf{Init}(\lambda)$ *is an algorithm that takes as input a public parameter* λ*, and outputs an empty append-only data store* ADS*.*
- $v \leftarrow \mathsf{Get}(\mathsf{ADS}, r)$ *is an algorithm that takes as input an append-only data store* ADS *and an address* r *and outputs a response* v *that corresponds to the value stored at address* r*.*
- $(\mathsf{ADS}', r) \leftarrow \mathsf{Put}(\mathsf{ADS}, v)$ *is an algorithm that takes as input an append-only data store* ADS *and a value* v*, and outputs an address* r *and an updated append-only data store* ADS'*.*

$\mathsf{ADS}[r]$ denotes the value stored at location r and $\mathsf{addr}(v)$ to denote the address at which v is stored in ADS.

5.4 LSX: A List-Based Scheme

In this section, we describe our first construction LSX. This is a multi-map encryption scheme that makes use of an append-only data store. There are two main technical challenges that occur when designing such a scheme. The first is handling delete operations since ADS's do not have the ability to modify or delete entries. The second is supporting the insertion of variable-length tuples efficiently. To solve these issues, we use three techniques: (1) *linking*, where we super-impose a linked list structure on top of the underlying ADS; (2) *lazy deletion*, where items are only marked for deletion at delete time and removed from the output at query time; and (3) *packing*, where we store multiple tuple values in one ADS entry.

Overview. At a high-level our scheme works as follows. Given a label/tuple pair that needs to be added or deleted, the client encrypts and stores the tuple values into the ADS maintained by the server. But, to differentiate between added and deleted values, it first concatenates a flag to each value: ADD for added values and $mathsfDEL$ for deleted values. At query time, the client reads all the values and outputs the ones that were added but not deleted.

Linking. Recall that in order to retrieve a value from an append-only data store ADS, one needs to know the address at which it is stored and this address cannot be computed or known a-priori by the client. To support search, a naive approach would be to require the client to store the addresses of all the values that it ever stored in the data store; which is obviously very space inefficient. To improve this, the client will super-impose in the ADS a virtual linked list over the values associated to a label ℓ and store locally only the address of the tail of the list. More precisely, it works as follows: for each label ℓ, and each value in ℓ's tuple, the client concatenates to the value the address of the previous value that was stored in ADS. It then stores the address of the last value in the tuple. Overall, the client only needs to keep a state that is linear in the number of labels in the multi-map. Notice that it is possible to achieve constant size state by super-imposing a single list

over the values of all labels but we dismissed this approach since the query time would be linear in the number of values of all labels.

Packing. Depending on how big the tuple values are, it is possible to pack multiple values in one entry of the data store. This trivially makes the construction more efficient since queries will require a fewer number of interactions with the ADS. Let λ be a data-store-specific parameter that denotes the maximum number of bits that can be stored in an entry of the ADS. The client then packs the maximum number of values in what we call a *value-block* such that the total size of the value-block (including the flag and the address) is at most λ. It then stores the value-block as an entry of the data store.

5.4.1 Details

The LSX scheme makes a black-box use of a private key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and an append-only data store $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$. LSX is described in detail in Figure 5.1 and we provide below a high level overview on how it works.

Init. During initialization, given a security parameter 1^k as input, the client generates an encryption key K and initializes a dictionary DX, while the server initializes its append-only data store ADS using $\Sigma_{\mathsf{ADS}}.\mathsf{Init}(1^k, \perp)$. The client uses dictionary DX to store addresses of the values it stores in ADS. More precisely, $\mathsf{ADS}[\ell]$ is the address of the last value stored in ADS associated with label ℓ.

Edit$^+$. To add a tuple $\mathbf{v}$ to an (exisiting) label ℓ, the client chops $\mathbf{v}$ into *value-blocks* $B^{(1)}, \dots B^{(t)}$, such that the size (in bits) of each encrypted value-block appended with an address and an ADD flag is at most λ. It then does the following for each $i \in [t]$: it first encrypts $(B^{(i)} \,||\, \mathsf{ADD} \,||\, r^{i-1})$, where $r^{(i-1)}$ is the address of the previous block stored in $\mathsf{ADS}^{(i-1)}$ for ℓ,[3] and then stores the encrypted value e in $\mathsf{ADS}^{(i-1)}$ and computes new address $r^{(i)}$ where $(r^i, \mathsf{ADS}^{(i)}) \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Put}(\mathsf{ADS}^{(i-1)}, e)$. The client finally updates $\mathsf{DX}[\ell] = r^{(|\mathbf{v}|)}$, so that it can correctly extend the chain on next $\mathsf{Edit}^+/\mathsf{Edit}^-$ operation.

Edit$^-$. Deletion is same as addition with the difference that value-blocks are now concatenated with $mathsfDEL$ flag instead of ADD flag.

Query. To compute $\mathsf{MM}[\ell]$, the client sends to the server the address $r = \mathsf{DX}[\ell]$, which the server uses to retrieve and return $e = \mathsf{ADS}[r]$. The client then decrypts e to recover a value block B, a flag flag and an address r'. If $\mathsf{flag} = \mathsf{ADD}$, it adds B to a set V or else it adds it to V_d. It then sets $r = r'$ and checks if $r = \perp$. If $r = \perp$, it has retrieved all the values that were ever added/deleted

[3] Note that the first added value of any label ℓ is concatenated with $\perp$ in order to be able to identify the end of a label's chain.

to/from ℓ, and if not, it repeats. Finally, it outputs the set of values in $V \setminus V_d$, which intuitively represent the values that were added but not yet deleted.

Let $\lambda \geq 0$ be a public parameter, let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ be an append-only data store. Consider the dynamic encrypted multi-map $\mathsf{LSX} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Edit}^+, \mathsf{Edit}^-)$ defined as follows:

- $\mathsf{Init}(1^k; \perp)$:
  1. $\mathbf{C}$ generates $K \leftarrow \mathsf{SKE.Gen}(1^k)$;
  2. $\mathbf{C}$ initializes an empty dictionary DX;
  3. $\mathbf{S}$ initializes an empty append-only data store $\mathsf{ADS} \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Init}(\lambda)$;
  4. $\mathbf{C}$ outputs a state $st = \mathsf{DX}$ and a key K, whereas $\mathbf{S}$ outputs ADS.

- $\mathsf{Edit}^+(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as DX and sets ADS to $\mathsf{ADS}^{(0)}$;
  2. $\mathbf{C}$ sets $r^{(0)} \leftarrow \mathsf{DX}[\ell]$;
  3. $\mathbf{C}$ chops $\mathbf{v}$ into maximal sized value-blocks $B^{(1)}, \ldots, B^{(t)}$ such that for each $i \in [t]$, $|\mathsf{SKE.Enc}_K(B^{(i)} \mid\mid r^{(0)} \mid\mid \mathsf{ADD})| \leq \lambda$;
  4. for each $i \in [t]$:
     (a) $\mathbf{C}$ computes $e \leftarrow \mathsf{SKE.Enc}_K(B^{(i)} \mid\mid r^{(i-1)} \mid\mid \mathsf{ADD})$;
     (b) $\mathbf{C}$ sends e to the server $\mathbf{S}$;
     (c) $\mathbf{S}$ computes $(\mathsf{ADS}^{(i)}, r^{(i)}) \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Put}(\mathsf{ADS}^{(i-1)}, e)$;
     (d) $\mathbf{S}$ returns $r^{(i)}$ to $\mathbf{C}$;
  5. $\mathbf{C}$ sets $\mathsf{DX}[\ell] = r^{(t)}$.

- $\mathsf{Edit}^-(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. It is the same as Edit^+, except that at line 3a, $\mathbf{C}$ concatenates the $mathsfDEL$ flag instead of ADD flag.

- $\mathsf{Query}(st, K, \ell; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as DX;
  2. $\mathbf{C}$ sets $r = \mathsf{DX}[\ell]$ and initializes two empty sets V and V_d;
  3. while $r \neq \perp$,
     (a) $\mathbf{C}$ sends r to $\mathbf{S}$;
     (b) $\mathbf{S}$ computes $e \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Get}(\mathsf{ADS}, r)$ and sends e to $\mathbf{C}$;
     (c) $\mathbf{C}$ computes $(B \mid\mid r' \mid\mid \mathsf{flag}) \leftarrow \mathsf{SKE.Dec}_K(e)$;
     (d) if $\mathsf{flag} = \mathsf{ADD}$, $\mathbf{C}$ appends B to V;
     (e) otherwise if $\mathsf{flag} = mathsfDEL$, $\mathbf{C}$ appends B to V_d;
     (f) $\mathbf{C}$ sets $r = r'$.
  4. $\mathbf{C}$ outputs $V \setminus V_d$.

Figure 5.1: The LSX scheme.

Security. We now describe the leakage profile of the LSX scheme. The initialization leakage is equal to

$$\mathcal{L}_{\mathsf{I}}(\perp) = \perp.$$

The query leakage is equal to

$$\mathcal{L}_{\mathsf{Q}}(\mathsf{MM}, \ell) = \mathsf{ueq}(\ell),$$

where $\mathsf{ueq}(\ell)$ is the *update equality pattern* which reveals if and when the label ℓ edit had occurred. More formally, it is defined as a bit string of length equal to the number of operations performed until now where the i^{th} bit is set to 1 if the i^{th} operation was an $\mathsf{Edit}^+/\mathsf{Edit}^-$ on ℓ, and 0 otherwise.

The add/delete leakage is equal

$$\mathcal{L}_{\mathsf{E}^+}(\mathsf{MM}, (\mathsf{Edit}^+, \ell, \mathbf{v})) = \mathcal{L}_{\mathsf{E}^-}(\mathsf{MM}, (\mathsf{Edit}^-, \ell, \mathbf{v})) = |\mathbf{v}|,$$

where $|\mathbf{v}|$ denotes the number of values being added/deleted from the multi-map.

Theorem 5.4.1. *If* SKE *is RCPA secure, then* LSX *is a* $(\mathcal{L}_{\mathsf{I}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}^+}, \mathcal{L}_{\mathsf{E}^-})$*-secure multi-map encryption scheme.*

Proof. Consider the simulator Sim that works as follows. It simulates the adversary $\mathcal{A}$ and first generates a symmetric key $K \leftarrow \mathsf{SKE.Gen}(1^k)$.

- Simulating Edit^+: On receiving $\mathcal{L}_{\mathsf{E}^+}(\mathsf{MM}, \ell, \mathbf{v}) = |\mathbf{v}|$, Sim sets $r^{(0)} = \perp$, creates a vector $\overline{\mathbf{v}}$ with random $|\mathbf{v}|$ values, chops $\overline{\mathbf{v}}$ into maximal sized value-blocks $\overline{B}^{(1)}, \ldots, \overline{B}^{(t)}$ such that for each $j \in [t]$, $|\mathsf{SKE.Enc}(\overline{B}^{(j)} \mid\mid r^{(0)} \mid\mid \mathsf{ADD})| \leq \lambda$, and repeats the following t times, i.e. for $i \in [t]$, it generates $e_i \leftarrow \mathsf{SKE.Enc}_K(\overline{B}^{(i)} \mid\mid r^{(i-1)} \mid\mid \mathsf{ADD})$, sends e_i to $\mathcal{A}$, waits for $\mathcal{A}$ to return a new $r^{(i)}$ and then it repeats. It also associates and stores $r^{(t)}$ with current time.

- Simulating Edit^-: It is exactly same as simulation of Edit^+.

- Simulating Query: Given $\mathcal{L}_{\mathsf{Q}}(\mathsf{MM}, \ell) = \mathsf{ueq}(\ell)$, Sim first sorts the update times in descending order and stores the sorted times in tuple $\mathbf{u}$. For each $u \in \mathbf{u}$, it then uses the address r associated with u, sends r to $\mathcal{A}$, waits for $\mathcal{A}$ to return e, decrypts e to compute next r and repeats until $r = \perp$.

It remains to show that for all PPT adversaries $\mathcal{A}$, the probability that $\mathbf{Real}(k)$ outputs 1 is negligibly close to the probability that $\mathbf{Ideal}(k)$ outputs 1. This can be done with the following sequence of games:

Game_0 : is the same as a $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ experiment.

Game_1 : is the same as Game_0 except that the encryptions of $(B \mid\mid r \mid\mid \mathsf{flag})$ during Edit^+ and Edit^- are replaced by encryptions of $(\overline{B} \mid\mid r \mid\mid \mathsf{flag})$, where $\overline{B}$ is created by chopping a vector $\overline{\mathbf{v}}$ of random values.

Game_2 : is the same as Game_1 except for the following. On an Edit^+ or Edit^-, we initialize $r^{(0)}$ to $\perp$ instead of initializing it to $\mathsf{DX}[\ell]$. We also associate and store $r^{(t)}$ with the current time. Then, on a query, we sort the update times of ℓ, and use the addresses associated with the sorted update times to query the server.

Note that Game_0 and Game_1 are indistinguishable because otherwise the encryption scheme is not RCPA secure – on Edit^+ and Edit^-, the vector $\mathbf{v}$ of Game_0 is replaced with a random vector $\overline{\mathbf{v}}$ in Game_1. Game_1 and Game_2 are also indistinguishable because the encryption is RCPA secure – on Edit^+ and Edit^-, $r^{(0)}$ of Game_1 is replaced with $\perp$ in Game_2. The Query protocol remains the same from the adversary's perspective as it receives the same set of r values as it would in Game_1. Proof concludes by noticing that Game_2 is $\mathbf{Ideal}(k)$ experiment.

■

Efficiency. We evaluate our scheme (as well as the next two schemes) based on three parameters: (1) time complexity, time, which is the amount of work done by the server; (2) round complexity for reads, $\mathsf{rounds_r}$, which is the number of communication rounds that take place between the client and the server for reads; and (3) round complexity for writes, $\mathsf{rounds_w}$, which is the number of communication rounds that take place between the client and the server for writes. We evaluate the round complexity of our schemes separately for reads and writes because when the underlying ADS is instantiated with a blockchain, writes can take much longer than reads. When the ADS is a blockchain, we sometimes use the term *stabilization complexity*, which we denote stbl, to refer to the round complexity of writes. This is because the time it takes to write a value/transaction to a blockchain depends on the time it takes the transaction to become stable. We summarize in Table 5.1 the time and stabilization complexities of LSX along with our two protocols described in the subsequent sections, and we give a detailed analysis below.

- Query. Since the query protocol requires reading all the value-blocks that were ever added or deleted from ℓ, its time complexity time is $O(|\mathbf{u}|)$, where $\mathbf{u}$ is the sequence of all update operations. The round complexity for reads $\mathsf{rounds_r}$ is $O(u)$ where $u = \sum_{i=1}^{|\mathbf{u}|} |\mathbf{v}_i|/\lambda$, where $\mathbf{v}_i$ is the tuple to the ith update operation in $\mathbf{u}$. This holds as the the address of the value-block $B^{(i)}$ cannot be computed until the value-block $B^{(i+1)}$ is read. Note that since nothing is written to the server during query time, the stabilization complexity stbl is not relevant.
- Edit^+ and Edit^-. Since $|\mathbf{v}|$ values are written in total, the time complexity is equal to $O(|\mathbf{v}|)$ independently of the packing factor. The stabilization complexity, however, i $O(|\mathbf{v}|/\lambda)$ since the value-block $B^{(i)}$ cannot be written unless $B^{(i-1)}$ has been written. Since nothing is read from the server, $\mathsf{rounds_r}$ is not relevant.

	$\mathsf{Edit}^+(\ell, \mathbf{v}_i)$		$\mathsf{Query}(\ell)$		$\mathsf{Edit}^-(\ell, \mathbf{v}_i)$		
	time	$\mathsf{rounds_w}$	time	$\mathsf{rounds_r}$	time	$\mathsf{rounds_r}$	$\mathsf{rounds_w}$
LSX	$O(\lvert\mathbf{v}_i\rvert)$	$O(\lvert\mathbf{v}_i\rvert/\lambda)$	$O(\lvert\mathbf{u}_i\rvert)$	$O(u_i)$	$O(\lvert\mathbf{v}_i\rvert)$	-	$O(\lvert\mathbf{v}_i\rvert/\lambda)$
TRX	$O(\lvert\mathbf{v}_i\rvert)$	$O(\log(\lvert\mathbf{v}_i\rvert/\lambda))$	$O(\lvert\mathbf{u}_i\rvert)$	$O(t_i)$	$O(\lvert\mathbf{v}_i\rvert)$	-	$O(\log(\lvert\mathbf{v}_i\rvert/\lambda))$
PAX	$O(\lvert\mathbf{v}_i\rvert)$	$O(\lvert\mathbf{v}_i\rvert)$	$O(\lvert\mathsf{MM}^i[\ell]\rvert)$	$O(\lvert\mathsf{MM}^i[\ell]\rvert)$	$O(\lvert\mathsf{MM}^{i-1}[\ell]\rvert + \lvert\mathbf{v}_i\rvert \log(\lvert\mathsf{MM}^{i-1}[\ell]\rvert))$	$O(\lvert\mathsf{MM}^{i-1}[\ell]\rvert)$	$O(\log(\lvert\mathsf{MM}^{i-1}[\ell]\rvert))$

Table 5.1: We denote by time the time complexity, by $\mathsf{rounds_r}$ the round complexity for reads, and by $\mathsf{rounds_w}$ the round complexity for writes. We denote by $\lambda > 0$ the packing parameter. The efficiency complexities are calculated for the ith add/delete operation. The sequence of updates $\mathbf{u}_i$ is composed of i add/delete operations. We denote by $u_i = \sum_{j=1}^{|\mathbf{u}_i|} |\mathbf{v}_j|/\lambda$ and by $t_i = \sum_{j=1}^{|\mathbf{u}_i|} \log(|\mathbf{v}_j|/\lambda))$.

5.5 TRX: Improving Stabilization Complexity

The round complexity of LSX for writes is linear in the length of the inserted or deleted tuple which means that its stabilization complexity when instantiated over a blockchain is also linear. More precisely, will be $O(|\mathbf{v}|/\lambda)$, where $\mathbf{v}$ is the tuple to be added and λ is the size of an entry in the underlying ADS. As we will see in our evaluation section, from a practical standpoint this leads to a non-trivial bottleneck for latency. To address this we propose a new scheme, TRX, with write round complexity $O(\log(|\mathbf{v}|/\lambda))$ and the same time complexity and client storage as LSX.

Overview. Recall that the Edit^+ and Edit^- protocols in LSX append to each value-block the address of the previous value-block stored in the ADS. This means that a value-block cannot be stored until the address of the previous value-block is available or, in the context of a blockchain, stable. Since both Edit^+ and Edit^- protocols link $|\mathbf{v}|/\lambda$ value-blocks linearly, the client must wait for $|\mathbf{v}|/\lambda$ value-blocks in total to become stable. Therefore, the write round complexity is $O(|\mathbf{v}|/\lambda)$. In TRX, we modify the way the value-blocks are organized with the goal of reducing the number of addresses needed before storing a value-blocks. Instead of using a linked list, we super-impose a complete binary tree which allows us to parallelize the insertions of multiple value-blocks. This approach helps reduce the number of rounds required for writes and, in the context of blockchains, decrease the stabilization complexity to be logarithmic instead of linear in the size of the tuple.

The tree structure. The TRX scheme super-imposes a complete binary tree structure over the value-blocks of $\mathbf{v}$ so that all the nodes on a level can be inserted in parallel. This is possible since storing a value-block only requires knowing the address of its parent. To do this, TRX concatenates two addresses, lp and rp, to every value-block B of the tuple. Here, lp and rp are the addresses of other value-blocks that were stored before B, i.e., that are at a lower level in the tree. lp represents the address of B's left child and rp the address of its right child.

Note that the tree structure is only created for value-blocks that belong to the same update operation. A valid question is how one could link value-blocks added across multiple Edit operations. For this, we simply link the roots of the trees together in a linear fashion similar to the LSX scheme.

5.5.1 Details

The TRX scheme makes black-box use of a private key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and an append-only data store $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$. TRX is described in detail in Figure 5.2 and we provide a high level overview of how it works below.

Init. The initialization protocol is similar to the one of LSX.

Edit$^+$. To add the tuple $\mathbf{v}$ to a label ℓ, the client first creates its value-blocks $B^{(1)}, \ldots, B^{(t)}$. For each $B^{(i)}$, the client then sets its lp to be the address of $B^{(2i)}$ and its rp to be the address of $B^{(2i+1)}$. The value-blocks on the last level of the logical tree have their lp and rp set to $\perp$. More precisely value-blocks $B^{(\lceil (t/2)+1)\rceil}$ to $B^{(t)}$ have their lp and rp set to $\perp$. Notice that this creates a tree structure among the value-blocks of $\mathbf{v}$. The client starts by storing value-blocks of $\mathbf{v}$ in reverse order so that when it stores $B^{(i)}$, the addresses of $B^{(2i)}$ and $B^{(2i+1)}$ have already been obtained. As before, it appends an ADD flag, encrypts $(B^{(i)} \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, \mathsf{ADD})$, and sends the encryption $e^{(i)}$ to the server. The server stores $e^{(i)}$ and returns the address $r^{(i)}$ back to the client. In order to link different tree structures belonging to the same label ℓ, the client also appends the address of the root of the last tree to the value-block stored at the root of the current tree. More precisely, the client also concatenates $\mathsf{DX}[\ell]$ to $(B^{(1)} \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, \mathsf{ADD})$ before encryption, refer to line 5b in Figure 5.2. Finally, the client updates $\mathsf{DX}[\ell]$ with the address of the root of the current tree.

Edit$^-$. This protocol is the same as Edit$^+$ with the difference that value-blocks are now concatenated with a $mathsfDEL$ flag instead of an ADD flag.

Query. The Query protocol is similar to the protocol of LSX with the difference that the client now traverses multiple trees. It sends to the server $r_{\mathsf{root}} = \mathsf{DX}[\ell]$ which is the root of the last tree stored in ADS. When the server returns $e = \mathsf{ADS}[r_{\mathsf{root}}]$, the client decrypts it to retrieve the address r'_{root} of the root of the next tree and the addresses of the two child nodes. It puts the addresses of the children on a stack S and uses the stack to do a depth-first search on the tree to retrieve all the value-blocks stored in that tree; refer to line 2h to 2(h)vi in Figure 5.2. As in LSX, if the retrieved value-block has an ADD flag, the client adds the value-block to a set V If not, the client adds it a set V_d. Finally it outputs $V \setminus V_d$.

Security. Since the only difference between TRX and LSX is how they represent the values logically in ADS (LSX represents them as a list whereas TRX represents them as list of trees) their leakage profiles as and their security proofs are the same. We therefore simply state the security theorem without giving its proof.

Theorem 5.5.1. *If* SKE *is RCPA-secure, then* TRX *is* $(\mathcal{L}_{\mathsf{I}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}^+}, \mathcal{L}_{\mathsf{E}^-})$*-secure multi-map encryption scheme.*

Let $\lambda \geq 0$ be a public parameter, let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ be an append-only data store. Consider the dynamic encrypted multi-map $\mathsf{LSX} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Edit}^+, \mathsf{Edit}^-)$ defined as follows:

- $\mathsf{Init}(1^k; \perp)$: same as Init protocol in Figure 5.1.
- $\mathsf{Edit}^+(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as DX and sets ADS as $\mathsf{ADS}^{(0)}$;
  2. $\mathbf{C}$ chops $\mathbf{v}$ into maximal sized value-blocks $B^{(1)}, \ldots, B^{(t)}$ such that for each $j \in [t]$, $|\mathsf{SKE.Enc}_K(B^{(j)} \,||\, r^{(0)} \,||\, r^{(0)} \,||\, \mathsf{ADD})| \leq \lambda$;
  3. $\mathbf{C}$ creates a new vector $\mathbf{r}$ of size $(2t+1)$ where $r^{(i)}$ denotes the address of $B^{(i)}$;
  4. $\mathbf{C}$ initializes all $r^{(i)}$ in $\mathbf{r}$ to $\perp$;
  5. for each i from t to 1,
     (a) $\mathbf{C}$ sets $\mathsf{lp} = r^{(2i)}$ and $\mathsf{rp} = r^{(2i+1)}$;
     (b) if $i = 1$, $\mathbf{C}$ computes and sends to $\mathbf{S}$
     $$e^{(i)} \leftarrow \mathsf{SKE.Enc}_K(B^{(i)} \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, \mathsf{DX}[\ell] \,||\, \mathsf{ADD});$$
     (c) otherwise if $i \neq 1$, $\mathbf{C}$ computes and sends to $\mathbf{S}$
     $$e^{(i)} \leftarrow \mathsf{SKE.Enc}_K(B^{(i)} \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, \mathsf{ADD});$$
     (d) $\mathbf{S}$ computes $(\mathsf{ADS}^{(i)}, r^{(i)}) \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Put}(\mathsf{ADS}^{(i-1)}, e^{(i)})$;
     (e) $\mathbf{S}$ returns $r^{(i)}$ to $\mathbf{C}$ who updates $\mathbf{r}$;
  6. $\mathbf{C}$ sets $\mathsf{DX}[\ell] = r^{(1)}$.
- $\mathsf{Edit}^-(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. It is the same as Edit^+, except that at lines 5b and 5c, $\mathbf{C}$ concatenates the $mathsfDEL$ flag instead of the ADD flag.
- $\mathsf{Query}(st, K, \ell; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as DX and sets $r_{\mathsf{root}} = \mathsf{DX}[\ell]$;
  2. while $r_{\mathsf{root}} \neq \perp$,
     (a) $\mathbf{C}$ sends r_{root} to $\mathbf{S}$;
     (b) $\mathbf{S}$ computes and sends $e \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Get}(\mathsf{ADS}, r_{\mathsf{root}})$ to $\mathbf{C}$;
     (c) $\mathbf{C}$ computes $(B \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, r'_{\mathsf{root}} \,||\, \mathsf{flag}) \leftarrow \mathsf{SKE.Dec}_K(e)$;
     (d) if $\mathsf{flag} = \mathsf{ADD}$, then $\mathbf{C}$ appends B to V;
     (e) otherwise if $\mathsf{flag} = mathsfDEL$, then $\mathbf{C}$ appends B to V_d;
     (f) $\mathbf{C}$ sets $r_{\mathsf{root}} = r'_{\mathsf{root}}$;
     (g) $\mathbf{C}$ initializes a stack S and pushes lp and rp in it;
     (h) while S is not empty,
        i. $\mathbf{C}$ sends $r \leftarrow S.\mathsf{pop}()$ to $\mathbf{S}$;
        ii. $\mathbf{S}$ computes and sends $e \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Get}(\mathsf{ADS}, r)$ to $\mathbf{C}$;
        iii. $\mathbf{C}$ computes $(B \,||\, \mathsf{lp} \,||\, \mathsf{rp} \,||\, \mathsf{flag}) \leftarrow \mathsf{SKE.Dec}_K(e)$
        iv. if $\mathsf{flag} = \mathsf{ADD}$, then $\mathbf{C}$ appends B to V;
        v. otherwise if $\mathsf{flag} = mathsfDEL$, then $\mathbf{C}$ appends B to V_d;
        vi. $\mathbf{C}$ computes $S.\mathsf{push}(\mathsf{lp})$ and $S.\mathsf{push}(\mathsf{rp})$ if they are not equal to $\perp$;
  3. $\mathbf{C}$ outputs $V \setminus V_d$.

Figure 5.2: The TRX scheme.

Efficiency. The efficiency of LSX is summarized in Table 5.1. We give a detailed analysis below.

- Edit^+ and Edit^-. Since $|\mathbf{v}|$ values are written in total, the time complexity time is $O(|\mathbf{v}|)$, independently of the packing factor λ. However, the stabilization complexity stbl is equal to $O(\log(|\mathbf{v}|/\lambda))$ because all value-blocks at the same level of the logical tree can be written in parallel.

- Query. Since the query protocol requires reading all the value-blocks that were ever added or deleted from ℓ, its time complexity time is $O(|\mathbf{u}|)$, where $\mathbf{u}$ is the sequence of all update operations. The round complexity for reads is equal to $t = \sum_{i=1}^{|\mathbf{u}|} \log(|\mathbf{v}_i|/\lambda)$, where $\mathbf{v}_i$ is the tuple of the ith update operation. Note that this holds true as all value-blocks at the same level can be read in parallel. Moreover, since nothing is written to the server during query time, the stabilization complexity stbl is not relevant.

5.6 PAX: Improving Query Efficiency

Both LSX and TRX have time complexities that are linear in the number of updates $|\mathbf{u}|$ ever made to a label ℓ; including the delete operations. This is because both schemes use lazy deletion with no rebuilds protocol as is the case for [60, 26]. As an example, if after $|\mathbf{u}|$ updates the client deletes all but one value of a tuple, the client and server still need to do a linear amount of work in the number of updates. In this section, we describe a new scheme, PAX, that achieves optimal time complexity at the cost of making delete operations slightly more expensive. PAX is based on a novel technique we call *patching*.

Overview. Like LSX, PAX stores added values in a single list. Unlike LSX, however, it does not use packing. Generalizing PAX to work support packing is left as open problem. At a high level, when a delete operation is executed, a set of *patches* are created and stored in the ADS. A patch is a pair of addresses s and d that will help traverse the lists without reading the deleted values. Now, a query operation will use the set of patches to skip over the deleted values and only read the required values. It is important to note, however, that to achieve time optimality, the number of patches has to be smaller than the number of values left. Achieving this is non-trivial, however, and requires us to organize the patches themselves in a tree structure.

5.6.1 Overview of Patching

A patch is a pair of addresses s and d, denoted $(s \to d)$, where s is the starting address and d the destination address. We first explain how delete operations trigger the creation of patches and then how the query operations use them. As a first step, assume that the client stores all the patches locally in a dictionary $\mathsf{DX}_{\mathsf{PT}}$. In this case, a patch $(s \to p)$ is stored as $\mathsf{DX}_{\mathsf{PT}}[s] = p$. We later explain how, instead of storing them locally, they can be stored in the ADS.

Patch creation. Each value v stored in the ADS has an implicit *predecessor* address and an implicit *successor* address. More formally, the predecessor is the address of the value that was added before v and is not yet deleted while the successor is the address of the value that was added after v and is not yet deleted. When v is deleted, a patch $(\mathsf{succ}(v) \rightarrow \mathsf{pred}(v))$ is created, where $\mathsf{succ}(v)$ is the address of v's successor and $\mathsf{pred}(v)$ is the address of v's predecessor.

Querying using patches. Querying with patching works as follows. Upon querying $\mathsf{ADS}[\mathsf{succ}(v)]$, the client recovers $\mathsf{addr}(v)$ and is then supposed to query the ADS for v. With patching, however, it can check its local dictionary $\mathsf{DX}_{\mathsf{PT}}$ to see if a patch $\big(\mathsf{succ}(v) \rightarrow \mathsf{pred}(v)\big)$ exists. If so, it can skip querying the ADS on $\mathsf{addr}(v)$ and directly jump to querying on $\mathsf{pred}(v)$. Intuitively, a patch $\big(\mathsf{succ}(v) \rightarrow \mathsf{pred}(v)\big)$ provides a way to retrieve $\mathsf{pred}(v)$ without reading $\mathsf{addr}(v)$.

Storing patches. Of course, storing patches locally would require too much client storage. In fact, as we will see, the number of patches needed in the worst-case is size of the entire encrypted multi-map. Fortunately, we can overcome this by storing the patches in the ADS. The patches will themselves need to be linked (so that we can find them) but, unlike the tuple values, they will be linked using a tree structure (the reason will be explained below). To create a tree structure, we concatenate two addresses lp and rp to every patch $p = (s \rightarrow d)$, where lp and rp are addresses of other patches that were stored before p. lp is the address of the left child and rp the address of the right child. The keys of the binary search tree are the starting address of the patches, i.e., the order of the patches is determined by their starting addresses s. We refer to this tree as the *patch tree*. As is done for the linked list structures, the client stores the address of the patch at the root of the tree.

Storing a new patch. When we store a new patch $P = (s \rightarrow p)$ in the data store, we need to make sure that we maintain the virtual binary search tree. This can be done using the following steps:

- *(patch creation).* Since P is a new patch, it is going to be a leaf in the tree. The client first concatenates to P its two child pointers $\mathsf{lp} = \perp$ and $\mathsf{rp} = \perp$, i.e., it has the following form

  $$P = \Big((s \rightarrow p) \mid\mid \perp \mid\mid \perp\Big).$$

  It then sends P to the server who stores it in ADS and returns its address r_P to the client.

- *(patch position).* To find the patch's insertion location in the tree, the client sends to the server the address of the root of the patch tree. The server reads and returns the patch $P_{\mathsf{root}} = (s_{\mathsf{root}} \rightarrow \mathsf{P}_{\mathsf{root}}) \mid\mid \mathsf{lp}_{\mathsf{root}} \mid\mid \mathsf{rp}_{\mathsf{root}}$. The client then checks if $s < s_{\mathsf{root}}$. If so, it sends $\mathsf{lp}_{\mathsf{root}}$ to the server otherwise it sends $\mathsf{rp}_{\mathsf{root}}$. The client continues this process until it retrieves a patch

  $$P_{\mathsf{parent}} = (s_{\mathsf{parent}} \rightarrow p_{\mathsf{parent}}) \mid\mid \mathsf{lp}_{\mathsf{parent}} \mid\mid \mathsf{rp}_{\mathsf{parent}}$$

  where P needs to be inserted.

- *(path modification).* At this point, the normal binary search tree insertion operation just involves changing one of the two addresses $\mathsf{lp}_{\mathsf{parent}}$ or $\mathsf{rp}_{\mathsf{parent}}$ to the address r_P of the parent P. However, since P_{parent} is stored in an append-only data store, P_{parent} cannot be changed. The client, therefore, creates a new node P'_{parent}, sets the patch content in P'_{parent} to be the same as the patch content in P_{parent}, changes one of the addresses $\mathsf{lp}_{\mathsf{parent}}$ or $\mathsf{rp}_{\mathsf{parent}}$ to r_P. It then sends P'_{parent} to the server who stores it in ADS and returns an address $r_{P'_{\mathsf{parent}}}$. Unfortunately this requires an address in parent of P_{parent} to be changed to $r_{P'_{\mathsf{parent}}}$, which cannot be done since it is stored in append-only data store. So a new node for the parent of P_{parent} is created. This process propagates up to the root and every node on the path from P to the root is replaced by a new node. When the server returns the address of the new root node, the client updates its copy of the root address.

Also, since every addition to the patch tree triggers the creation of as many nodes as its height, we keep the tree balanced for efficiency reasons. Balancing the tree follows the same approach detailed above so we omit the details from this version of the paper.

Cleaning up a patch tree. Cleaning up the patch tree is the process of deleting some patches to achieve both time optimality and correctness. Let us start with an example to illustrate when and why a cleanup might be required. Assume that the unencrypted contents of the data store ADS are: $\mathsf{ADS}[r_1] = (v_1 \,||\, \perp)$, $\mathsf{ADS}[r_2] = (v_2 \,||\, r_1)$, $\mathsf{ADS}[r_3] = (v_3 \,||\, r_2)$, and $\mathsf{ADS}[r_4] = (v_4 \,||\, r_3)$. Then, when v_3 is deleted, a patch $(r_4 \rightarrow r_2)$ is created. However, if v_2 is deleted next, the creation of a new patch $(r_4 \rightarrow r_1)$ means we have two patches with the same starting address r_4. It is therefore not clear which patch is the right one. Moreover, the number of patches in the patch tree will then be equal to the number of values deleted, which means that searching for a patch in the patch tree is as expensive as employing the lazy deletion approach. To avoid these issues, we need a way to delete old patches. In the example above, suppose we want to delete $r_4 \rightarrow r_2$ and add $r_4 \rightarrow r_1$. Deletion from a patch tree is similar to an addition: we first find the node P to be deleted, replace it with the appropriate descendant node P' by replacing nodes on path from P to the root with new nodes with appropriate pointer changes.

5.6.2 Details

The PAX scheme makes black-box use of a private key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and an append-only data store $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$. PAX is described in detail in Figure 5.3 and we provide below a high level overview of how it works.

Init. The initialization of PAX is similar to the one of LSX with the difference that now the client also initializes an empty dictionary $\mathsf{DX}_{\mathsf{Root}}$ to store the addresses of the roots of the patch trees.

Edit$^+$. Edit^+ is also similar to the one of LSX but with the following differences: (1) it no longer concatenates the ADD flag to the values that it stores in ADS; (2) it no longer packs values in

value-blocks; and (3) it stores the values in $\mathbf{v}$ in a random order in ADS.

Query. As a first step, the client reads all the patches in the patch tree using the root of the patch tree $\mathsf{DX}_{\mathsf{Root}}[\ell]$. It stores them in a local dictionary $\mathsf{DX}_{\mathsf{PT}}$ by storing a patch $(s \to d)$ as $\mathsf{DX}_{\mathsf{PT}}[s] = d$. To compute $\mathsf{MM}[\ell]$, it sends the address $r = \mathsf{DX}[\ell]$ to the server who uses it to retrieve and return $e = \mathsf{ADS}[r]$. The client decrypts e to recover a value v and an address r'. It adds v to a set V and checks if there exists a patch starting at r by checking if there exists an entry corresponding to r in $\mathsf{DX}_{\mathsf{PT}}$. If so, it sets $r = \mathsf{DX}_{\mathsf{PT}}[r]$, else it sets $r = r'$. It repeats this process until $r = \perp$, at which point it has retrieved all the values that are currently in $\mathsf{MM}[\ell]$. Finally it outputs the set V.

Edit$^-$. As first step, the client reads all the patches and stores them locally in $\mathsf{DX}_{\mathsf{PT}}$. It then starts reading all the values stored in ADS one-by-one. While reading, it also skips over the deleted values using the downloaded patches as it does during query. The aim of this traversal is to compute patches for values that are to be deleted. Recall that a patch for a value is its successor and predecessor addresses. Therefore, during traversal, when the client reads a value $v \in \mathbf{v}$, it stores $(\mathsf{succ}(v) \to \mathsf{pred}(v))$ in a set P and $\mathsf{addr}(v)$ in a set C. There is a subtlety here though: the client has to be careful when creating these successor-predecessor pairs. For example, if two consecutive values are being deleted then they should share the same successor-predecessor pair. Once the client collects all the new patches in P, it needs to add all of them to the patch tree. But before it does so, it checks if there is some cleanup needed to the patch tree.

Security. We now describe the leakage profile of PAX. The initialization leakage is $\mathcal{L}_{\mathsf{I}}(\perp) = \perp$. The query leakage is

$$\mathcal{L}_{\mathsf{Q}}(\mathsf{MM}, \ell) = \mathsf{aeq}(\ell),$$

where aeq is the *add equality pattern* which reveals if and when additions were made to the label. The add leakage is

$$\mathcal{L}_{\mathsf{E}^+}(\mathsf{MM}, \ell, \mathbf{v}) = |\mathbf{v}|,$$

where $|\mathbf{v}|$ is the number of values being added. The delete leakage is

$$\mathcal{L}_{\mathsf{E}^-}(\mathsf{MM}, \ell, \mathbf{v}) = \big(\mathsf{aeq}(\ell), \{\mathsf{vad}(\ell, v)\}_{v \in \mathbf{v}}\big),$$

where aeq is the add equality pattern, and vad is the *value addition pattern* of all the values that are being deleted, where vad represents if and when a value was added to a label.

Theorem 5.6.1. *If* SKE *is RCPA-secure, then* PAX *is a* $(\mathcal{L}_{\mathsf{I}}, \mathcal{L}_{\mathsf{Q}}, \mathcal{L}_{\mathsf{E}^+}, \mathcal{L}_{\mathsf{E}^-})$*-secure multi-map encryption scheme.*

Proof. Consider the simulator Sim that works as follows. It simulates the adversary $\mathcal{A}$ and first generates a symmetric key $K \leftarrow \mathsf{SKE.Gen}(1^k)$.

- Simulating Edit^+: On receiving $\mathcal{L}_{\mathsf{E}^+}(\mathsf{MM}, \ell, \mathbf{v}) = |\mathbf{v}|$, Sim sets $r_0 = \perp$, and repeats the following $|\mathbf{v}|$ times: for $i \in [|\mathbf{v}|]$, it generates $e_i \leftarrow \mathsf{SKE.Enc}_K(0^{|v_i|} \,||\, r_{i-1})$, sends e_i to $\mathcal{A}$, waits for $\mathcal{A}$ to

Let $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme and $\Sigma_{\mathsf{ADS}} = (\mathsf{Init}, \mathsf{Get}, \mathsf{Put})$ be an append-only data store. Consider the dynamic encrypted multi-map $\mathsf{PAX} = (\mathsf{Init}, \mathsf{Query}, \mathsf{Edit}^+, \mathsf{Edit}^-)$ defined as follows:

- $\mathsf{Init}(1^k; \perp)$: Same as in Figure 5.1, with the difference that client also initializes an extra dictionary $\mathsf{DX}_{\mathsf{Root}}$.
- $\mathsf{Edit}^+(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as $(\mathsf{DX}, \mathsf{DX}_{\mathsf{Root}})$ and sets $r^{(0)} \leftarrow \mathsf{DX}[\ell]$
  2. $\mathbf{S}$ lets $\mathsf{ADS}^{(0)} = \mathsf{ADS}$;
  3. for each $i \in [|\mathbf{v}|]$ in random order,
     (a) $\mathbf{C}$ computes and sends $e \leftarrow \mathsf{SKE.Enc}_K(v^{(i)} \mid\mid r^{(i-1)})$ to $\mathbf{S}$;
     (b) $\mathbf{S}$ computes $(\mathsf{ADS}^{(i)}, r^{(i)}) = \Sigma_{\mathsf{ADS}}.\mathsf{Put}(\mathsf{ADS}^{(i-1)}, e)$ and sends $r^{(i)}$ to $\mathbf{C}$;
  4. $\mathbf{C}$ sets $\mathsf{DX}[\ell] = r^{(|\mathbf{v}|)}$.
- $\mathsf{Query}(st, K, \ell; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as $(\mathsf{DX}, \mathsf{DX}_{\mathsf{Root}})$;
  2. Starting from $\mathsf{DX}_{\mathsf{Root}}[\ell]$, $\mathbf{C}$ reads all patches and stores them into a local dictionary $\mathsf{DX}_{\mathsf{PT}}$, where patch $(s \to d)$ is stored as $\mathsf{DX}_{\mathsf{PT}}[s] = d$;
  3. $\mathbf{C}$ sets $r = \mathsf{DX}[\ell]$;
  4. while $r \neq \perp$,
     (a) $\mathbf{C}$ sends r to $\mathbf{S}$
     (b) $\mathbf{S}$ computes and sends $e \leftarrow \Sigma_{\mathsf{ADS}}.\mathsf{Get}(\mathsf{ADS}, r)$ to $\mathbf{C}$;
     (c) $\mathbf{C}$ computes $(v \mid\mid r') \leftarrow \mathsf{SKE.Dec}_K(e)$ and appends v to V;
     (d) if $r \in \mathsf{DX}_{\mathsf{PT}}$, then $\mathbf{C}$ sets $r = \mathsf{DX}_{\mathsf{PT}}[r]$
     (e) otherwise if $r \notin \mathsf{DX}_{\mathsf{PT}}$, then $\mathbf{C}$ sets $r = r'$;
  5. $\mathbf{C}$ outputs V.
- $\mathsf{Edit}^-(st, K, \ell, \mathbf{v}; \mathsf{ADS})$:
  1. $\mathbf{C}$ parses st as $(\mathsf{DX}, \mathsf{DX}_{\mathsf{Root}})$;
  2. Starting from $\mathsf{DX}_{\mathsf{Root}}[\ell]$, $\mathbf{C}$ reads all patches and stores them into a local dictionary $\mathsf{DX}_{\mathsf{PT}}$, where patch $(s \to d)$ is stored as $\mathsf{DX}_{\mathsf{PT}}[s] = d$;
  3. Starting from $\mathsf{DX}[\ell]$, $\mathbf{C}$ reads all values in the chain and computes the following two sets,
     $$P = \{\mathsf{succ}(v) \to \mathsf{pred}(v) \mid v \in \mathbf{v}\} \quad C = \{\mathsf{addr}(v) \mid v \in \mathbf{v}\};$$
  4. for all $c \in C$,
     (a) if there exists a patch $(c \to d)$ in $\mathsf{DX}_{\mathsf{PT}}$, then $\mathbf{C}$ deletes $(c \to d)$ from $\mathsf{ADS}_{\mathsf{Root}}$;
  5. for all $(s \to p) \in P$:
     (a) if there exists a patch $(s \to p')$ in $\mathsf{DX}_{\mathsf{PT}}$, then $\mathbf{C}$ replaces $(s \to p')$ with $(s \to p)$ in $\mathsf{ADS}_{\mathsf{Root}}$;
     (b) otherwise, $\mathbf{C}$ adds $(s \to p)$ in $\mathsf{ADS}_{\mathsf{Root}}$.

Figure 5.3: The PAX scheme.

return a new r_i and then it continues. It also associates and stores $r_{|\mathbf{v}|}$ with the current time t. We denote $r_{|\mathbf{v}|}$ by head_t. Sim also stores all the addresses r_i in a set R_t.

- Simulating Edit^-: On receiving

$$\mathcal{L}_{\mathsf{E}^-}(\mathsf{MM}, \ell, \mathbf{v}) = (\mathsf{aeq}(\ell), \{\mathsf{vad}(\ell, v)\}_{v \in \mathbf{v}}),$$

  Sim first downloads the patch tree using the root stored in $\mathsf{DX}_{\mathsf{root}}[\min(\mathsf{aeq}(\ell))]$. Note that the simulator is only given as leakage the time $\mathsf{vad}(\ell, v)$ which is the time at which v was added. But there can be multiple values that were added at the same time as v. Therefore for each $t \in \{\mathsf{vad}(\ell, v)\}_{v \in \mathbf{v}}$, Sim selects $r_t \xleftarrow{\$} R_t$, adds it to a set C and removes r_t from R_t. Set C intuitively represents the set of addresses to be deleted. Sim then computes the predecessor and successor pairs of addresses in C by using heads in $\{\mathsf{head}_t \mid t \in \mathsf{aeq}(t)\}$ and the patch tree (this is similar to how P is computed in Line 3 of Figure 5.3). Sim then follows Lines 4 and 5 of Figure 5.3 to update the patch tree. Finally, it updates $\mathsf{DX}_{\mathsf{root}}[\min(\mathsf{aeq}(\ell))]$ to store the new root of the patch tree.

- Simulating Query: Given $\mathcal{L}_{\mathsf{Q}}(\mathsf{MM}, \ell) = \mathsf{aeq}(\ell)$, Sim first downloads the patch tree using the root in $\mathsf{DX}_{\mathsf{root}}[\min(\mathsf{aeq}(\ell))]$. It finally traverses the list by using heads in $\{\mathsf{head}_t \mid t \in \mathsf{aeq}(\ell)\}$ and the patch tree.

It remains to show that for all PPT adversaries $\mathcal{A}$, the probability that $\mathbf{Real}(k)$ outputs 1 is negligibly close to the probability that $\mathbf{Ideal}(k)$ outputs 1. This can be done with the following sequence of games:

Game_0 : is the same as a $\mathbf{Real}_{\mathcal{A},\mathcal{Z}}(k)$ experiment.

Game_1 : is the same as Game_0 except that the encryption of node $(v \mid\mid r)$ during Edit^+ and Edit^- is replaced by $\mathsf{SKE.Enc}_K(0^{|v|} \mid\mid r)$.

Game_2 : is the same as Game_1 except the following. On an Edit^+, we initialize r_0 to $\perp$ instead of initializing it to $\mathsf{DX}[\ell]$. We also set and store $\mathsf{head}_t = r_{|\mathbf{v}|}$ where t is the current time. Then, on Query and Edit^-, to read the chain of values, we sort the add equality pattern aeq of ℓ, and use head_t associated with $t \in \mathsf{aeq}(\ell)$ to read the values from the server. Moreover, on Edit^-, for all $v \in \mathbf{v}$, instead of deleting v, we select and delete an un-deleted random value from the set $\mathsf{vad}(\ell, v)$.

Note that Game_0 and Game_1 are indistinguishable because otherwise the encryption scheme is not RCPA-secure. Game_1 and Game_2 are also indistinguishable because the encryption is RCPA-secure – on Edit^+, r_0 of Game_1 is replaced with $\perp$ in Game_2. The Query protocol remains the same from the adversary's perspective as it receives the same set of r values as it would in Game_1. The Edit^- protocols are also indistinguishable because the Edit^+ protocol adds values in random order and thus the adversary cannot distinguish whether actually $v \in \mathbf{v}$ is getting deleted or some other value that was added at the same time as v. Proof concludes by noticing that Game_2 is $\mathbf{Ideal}(k)$ experiment.

■

Size of the patch tree. To assess the efficiency of PAX, we first need to bound the size of the patch tree. We do this in the following Theorem.

Theorem 5.6.2. *If T be the patch tree of a label ℓ, then $|T| \leq |\mathsf{MM}[\ell]|$.*

Proof. One can see by construction that for all patches $(s \to p)$ in T, if $v = \mathsf{ADS}[s]$ then $v \in \mathsf{MM}[\ell]$. Therefore the number of patches cannot be more than $|\mathsf{MM}[\ell]|$, provided that multiple patches starting at s are not stored in T. On the other had, notice that for any two patches $(s_1 \to p_1)$ and $(s_2 \to p_2)$, $s_1 \neq s_2$. That is, given an address s, there exists a unique patch that starts at s. Therefore, the number of patches is at most $|\mathsf{MM}[\ell]|$.

■

Efficiency. The efficiency of PAX is summarized in Table 5.1. We give a detailed analysis below.

- Edit^{+}. Since $|\mathbf{v}|$ values are written in total, the time complexity time is $O(|\mathbf{v}|)$. The stabilization complexity stbl is also $O(|\mathbf{v}|)$ because the ith value $v^{(i)}$ in $\mathbf{v}$ cannot be written unless $v^{(i-1)}$ has been already written.

- Query. Let $\mathsf{MM}^{(i-1)}$ be the state of the multi-map after the $(i-1)$th operation has been completed and let i be the current operation. Since the number of nodes in the patch tree are at most $|\mathsf{MM}^{i-1}[\ell]|$, refer to Theorem 5.6.2, it takes at most $|\mathsf{MM}^{i-1}[\ell]|$ time to download it. Once it is downloaded, finding patches onwards has a constant time in the size of the patch tree. Since the Query protocol only reads the un-deleted values, the time complexity time is $O(|\mathsf{MM}^{i-1}[\ell]|)$. The $\mathsf{rounds_r}$ is also $O(|\mathsf{MM}^{i-1}[\ell]|)$ because the un-deleted values can only be read sequentially.

- Edit^{-}. As explained for Query, downloading the patch structure and traversing the chain of un-deleted values to compute the new patches incurs a time and round complexity of $O(|\mathsf{MM}^{i-1}[\ell]|)$. There are no more reads that Edit^{-} does, therefore $\mathsf{rounds_r}$ is $O(|\mathsf{MM}^{i-1}[\ell]|)$. Since the patch tree is balanced, any addition/deletion to the tree updates constant number of logarithmic sized paths in the tree. Since there are at most $|\mathbf{v}|$ additions/deletions made, they together account for $O(|\mathbf{v}| \log(|\mathsf{MM}^{i-1}[\ell]|))$ time. Combining this time with the time taken by a traversal, the time complexity is $O(|\mathsf{MM}^{i-1}[\ell]| + |\mathbf{v}| \log(|\mathsf{MM}^{i-1}[\ell]|))$. Even though up to $|\mathbf{v}|$ logarithmic sized paths in the patch tree are updated, the updates to one level of the tree can be made in parallel. Therefore, the stabilization complexitystbl is $O(\log(|\mathsf{MM}^{i-1}[\ell]|))$.

5.7 Instantiating Append-only Data Stores with Blockchains

As discussed in Section 5.1, append-only data stores are an abstraction of blockchains and designing our schemes based on this abstraction means that our constructions can be used on any blockchain. The specifics of the underlying blockchain, however, have an impact on performance which we study

in Section 5.8. Here, we show concretely how two blockchains, Ethereum and Algorand, instantiate ADSs.

Overview. At a high level, an ADS can be instantiated with a blockchain as follows. The Init protocol creates a blockchain wallet for the client with some initial funds. The wallet also has a public address and a private key associated with it. The public address is only used to buy funds for the wallet while the private key is used to sign transactions. The $\mathsf{Put}(\mathsf{ADS}, v)$ protocol stores v in a transaction, signs it using the private key, and sends it to the blockchain. The address r of the value varies from one blockchain to another. For some, it is the transaction hash and for others it will be the transaction hash along with the transaction's block number after it is mined. The $\mathsf{Get}(\mathsf{ADS}, r)$ protocol communicates with one or more nodes to retrieve the transaction corresponding to address r and then retrieves the value v stored in that transaction.

We now discuss some practical implications of using a transaction hash over a block number (along with a transaction hash) as the address.

5.7.1 Instantiating Addresses

Both transaction hashes and block numbers (along with a transaction hash) are valid choices to instantiate ADS addresses. Which one should be used depends on: (1) how efficiently blockchain nodes create them at add time; and (2) how efficiently nodes can lookup transactions using them at query time.

Transaction hash. Using transaction hashes addresses can be very efficient at add time because these hashes can be computed locally by the client without needing to interact with the blockchain. Because of this, the client does not have to wait for older transactions to become stable before creating new ones. In this case, the stabilization complexity of LSX and TRX is independent of $|\mathbf{v}|$ which means that TRX's asymptotic advantage over LSX in terms of stabilization complexity disappears.

However, it is not always possible to use transaction hashes as addresses. In fact, due the storage overhead involved, many blockchains do not mandate that nodes support lookup by transaction hash. This is the case, for example, for Bitcoin, Ethereum, and Algorand. Of course some nodes—especially third party hosted nodes—might choose to implement lookup by transaction hash but this is not mandatory.

Block number. Using the block number along with the transaction hash as the address is a safer approach that guarantees that clients will be able to retrieve their data from the blockchain EMM. Unfortunately, using block numbers leads to higher stabilization complexity since the client has to wait for transactions to be mined and to become stable before it can use the address. For example, in Bitcoin, block numbers are only reliable after the block has reached a certain depth in the blockchain.

5.7.2 Using Ethereum

Ethereum is a proof-of-work based public blockchain that supports smart contracts. A smart contract is a program that is stored as a special transaction and executed by the blockchain. Each contract has some memory associated with it which is the state of the contract. The state can be changed by calling the functions of the program through transactions. The states of all contracts form the state of the blockchain which can then be seen as a *state transition machine*, where transactions stored in the blockchain specify how the state changes.

Details. We used Ethereum's `web3` API package [21] to interact with the Ethereum network. In particular, we instantiate each of the ADS protocols as follows:

- $\mathsf{Init}(\perp)$: We use Metamask [17] to create a wallet. Metamask automatically creates the public address and private key for the wallet. We then fund the wallet using the Metamask testnet faucet.

- $\mathsf{Put}(\mathsf{ADS}, v)$: We create and sign transactions using `web3`'s method call `signTransaction`(). `signTransaction` takes as input a dictionary with multiple fields (labels), one of which is called `data`. Traditionally, the `data` field is set to the bytecode of the function to be called followed by the function's arguments. However, it is also possible to provide custom input of up to 98KB. We use this field to store our values. `signTransaction` outputs a transaction that is signed by the private key but not yet submitted to the network. We then call `web3`'s `sendRawTransaction`() method which takes as input the signed transaction and sends it to the Ethereum network. It also outputs the hash r of the transaction which we use later to retrieve the transaction.

  To compute the block number, we execute the method call `waitForTransactionReceipt`() which takes as input a transaction hash, waits for the transaction specified by the hash to be mined, and returns the transaction's receipt. The receipt is an object which contains the block number in which the transaction is mined.

- $\mathsf{Get}(\mathsf{ADS}, r)$: We call `web3`'s `getTransaction`() method which takes as input the hash r of a transaction and outputs the associated transaction. We then read the `data` field to retrieve the value v.

  To retrieve the transaction by block number, we execute the `getBlock`() method, which takes as input the block number and outputs the block information. The block information contains a list of transactions which we scan to find our transaction.

5.7.3 Using Algorand

Algorand is a pure proof-of-stake public blockchain that provides high scalability and security without forking. We describe how an ADS can be implemented with Algorand. We used Algorand's `algosdk` [13] Python package for interacting with the Algorand network.

- $\mathsf{Init}(\perp)$: We use the `algosdk`'s `account.generate_account`() method to create a wallet. `generate_account` outputs a private key and account address for the wallet. We then fund the wallet using Algorand's testnet faucet.

- $\mathsf{Put}(\mathsf{ADS}, v)$: We create a transaction using the method called `transaction.PaymentTxn`(). It takes as input multiple parameters, one of which is called `note`. We write our data in the `note` field, which allows for up to 1KB of data. `PaymentTxn` outputs an unsigned transaction which we sign using the `sign`() method. We then call the `AlgodClient.send_transaction`() method which takes as input the signed transaction, sends it to the blockchain network and outputs the hash r of the transaction. To compute the block number containing the transaction with hash r, we call the `AlgodClient.transaction_info`() method. It returns multiple pieces of the confirmed transaction information, one of which is the block number.

- $\mathsf{Get}(\mathsf{ADS}, r)$: To retrieve the transaction by hash, we call the `AlgodClient.transaction_by_id`() method which takes as input the transaction hash r and outputs the corresponding transaction. We then read the `note` field and retrieve the value v. However, to retrieve the transaction by block number, we execute the `AlgodClient.block_info`() method, which takes as input the block number and outputs the block information. The block information contains a list of transactions which we scan to find our transaction.

5.8 Empirical Evaluation

To evaluate and compare the efficiency of our schemes, we implemented and evaluated them empirically. All the experiments were run on a MacBook Pro 2.8 GHz Intel Core i7 with 16GB of RAM. We implemented symmetric encryption with the `pycryptodome` library's AES implementation using a 128-bit key

Experimental setup for Ethereum. We used Metamask [17] to create a client wallet and connect it to a full node hosted by Infura [16]. We interact with the full node using Ethereum's `web3` API package [21] which is written in Python. Since running experiments on the mainnet is expensive, we did all of our experiments on Ethereum's Ropsten testnet and funded our wallet using Ropsten's faucet.

Experimental setup for Algorand. We used Algorand's `algosdk` [13] Python package to create a client wallet and connect it to a full node hosted by Purestake [20]. We interact with the full node using the same `algosdk` package. As for the Ethereum experiments, we ran them on Algorand's testnet and funded our wallet using the testnet's faucet.

Experimental data. We generated the experimental data synthetically. We created multi-maps that hold a single randomly-generated label/tuple pair. We created a single tuple because, for all our schemes, processing one label does not affect processing of any other labels. For example, in

LSX, the values associated with different labels are stored in different virtual lists so the query and update times for one label are not affected by query and update times of other labels. This is also true for TRX and PAX.

5.8.1 Experiments

We now describe our experiments and our findings. In all the experiments for LSX and TRX, we set λ to be 1KB for both Algorand and Ethereum.[4] We considered two ways to instantiate ADS addresses: (1) as transaction hashes; and (2) as block numbers (along with the transaction hash). Our goal was to evaluate the following characteristics of our schemes:

- *(add time):* time to add a label/tuple pair as a function of tuple size;
- *(delete time):* time to delete a label/tuple pair as a function of the tuple size. In particular, we are interested in three different delete modes: *sparse*, *dense*, and *random* deletes.
- *(query time):* time to query a label ℓ as a function of its tuple size. In particular, we measure the query time before and after delete operations.

Before describing the experiments, we discuss the problem of spam filtering that we faced during our experiments.

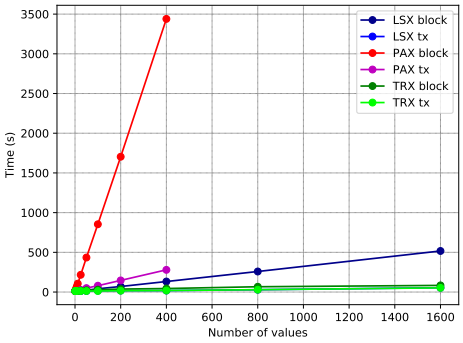
A note on DDoS protection. During our experiments on the Algorand testnet, our put and get requests failed every few operations which meant we had to wait for a non-deterministic amount of time before re-starting the experiments. We believe this occurred because the testnet is using a form of DDoS protection mechanism. The wait time was in seconds and is included in the times we report for both queries and updates.

We found that Ethereum was also employing a DDoS protection mechanism. In this case, however, the wait times were on the order of minutes which made our experiments infeasible.

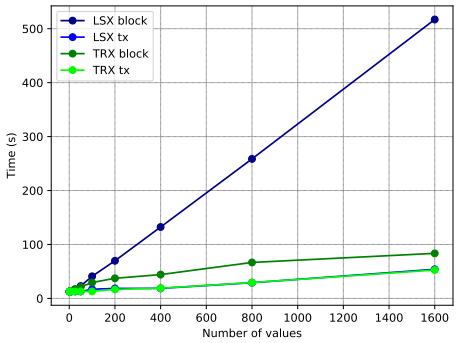
Measuring Edit$^+$. We measure the time our schemes take to add a label/tuple pair to the multi-map. For this, we create random label/tuple pairs $(\ell, \mathbf{v})$, with a number of values that increase from 1 to $1,500$. We then execute the Edit$^+$ to store them on the blockchain.

Figure 5.4 shows that instantiating addresses with block numbers (along with transaction hashes) is always more expensive than with transaction hashes. This is because transaction hashes can be computed locally even before the transaction is mined while block numbers are only known after the transaction is mined and can only be used after stabilization. We also noticed that TRX is much faster than LSX when we use block numbers as addresses. This is expected since TRX has logarithmic stabilization complexity while LSX has a linear stabilization complexity. When using transaction hashes, adding a label with a tuple of 400 values takes 18.53, 18.97, and 3416.56 seconds for LSX, TRX and PAX, respectively. PAX has the worst add time since it cannot pack multiple values in a transaction.

[4]Note that 1KB is the maximum packing factor for Algorand but not for Ethereum.



(a) Add complexity



(b) Add complexity (omitting PAX for legibility)

Figure 5.4: Edit$^+$ complexity for Algorand

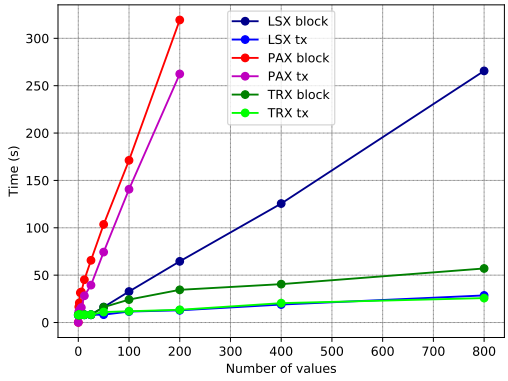
Measuring Edit$^-$. To measure deletion time, we considered three different patterns. For each one, we first store a label/tuple pair $(\ell, \mathbf{v})$ on the blockchain, then delete half of the values in the tuple. In the first mode, which we call *sparse deletes*, we delete every alternate value in $\mathbf{v}$. In the second mode, which we call *dense deletes*, we delete the first half of the values in $\mathbf{v}$. And in the third mode, which we call *random deletes*, we delete half of the values in $\mathbf{v}$ chosen uniformly at random.

Figure 5.5 shows that the delete time for LSX and TRX have a similar trend across all modes. They also have a similar trend as their add times (see Figure 5.4). This is expected since LSX and TRX need to add values during deletion. Moreover, we also see that the time LSX and TRX take to delete values is almost the same across all three modes which shows that their performance on deletes is independent of the mode. More precisely, to delete 200 values in the sparse setting, it takes 18.31 and 21.17 seconds for LSX and TRX.

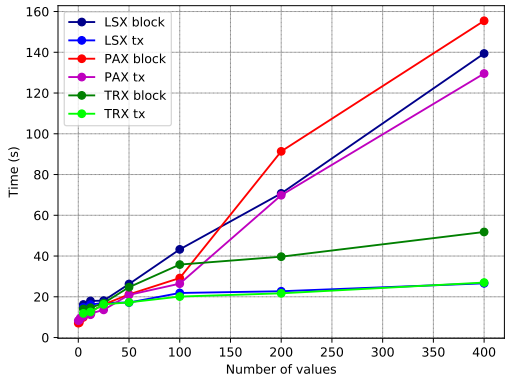
Finally, we see PAX takes longer than the other schemes for sparse deletes. This is expected since PAX spends a non-trivial amount of time creating and storing $|\mathbf{v}|/2$ patches in the patch tree. In the case of dense deletes, however, it only creates a single patch and we observe that PAX outperforms LSX and TRX when the number of values is smaller than 50 for the case of dense deletes.

Measuring Query. To measure the time to query a label, we conducted two experiments. The first measures query time after addition whereas the second measures query time after deletion. For the first experiment, we store a label/tuple pair and then query it. The second experiment consists of four sub-experiments, where we measure the query time after a sparse delete, a dense delete, a random delete, and an *all-but-one* delete. This last mode deletes all the values in the tuple except for one.

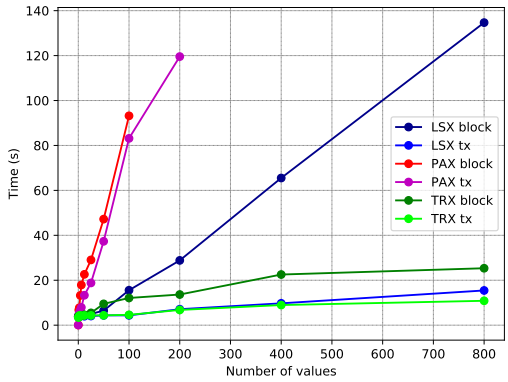
Figure 5.6 shows that using block numbers as addresses slows all the schemes down slightly compared to using transaction hashes. This is because, when block numbers are used, entire blocks need to be retrieved and scanned to find the transaction. This is clearly more expensive than retrieving the transaction directly. However, we did not notice a large gap between the two since



(a) Sparse deletes



(b) Dense deletes



(c) Random deletes

Figure 5.5: Edit^{-} complexity for Algorand

all the blocks contained only 1 or 2 transactions. Before deletions, the query time was 1.16, 1.08, 144.57 seconds for LSX, TRX and PAX, respectively. On the other hand, after dense deletes, PAX had slightly better query time of 66.19 seconds for 200 values. The query time for both LSX and TRX remains similar to the pre-deletion times. Finally, we observe that PAX outperforms the other two schemes when all the values of a tuple are deleted except for one. In this case, the query time is 0.64 seconds. This aligns with our theoretical results since the query protocol only needs to retrieve two transactions whereas it needs to retrieve hundreds of transactions in the other two schemes.

Block size. When the address is instantiated with block numbers, it is clear that the size of a block will affect the query time. We wanted to assess this impact experimentally but could not due to the anti-DDoS measures of the testnets so we carried out a simulation. Note that measuring the query time with a block size of 1 or 2 transactions and multiplying that by x to estimate the query time with block size of x or $2x$ would not work because our measured query times include wait times due to the anti-DDoS measures.

To address this, we carried out a separate experiment to estimate the average processing time of a block as a function of the number of transactions, where the processing time refers to the time it takes to download and scan the block to find the transaction, including the time wait time due to the anti-DDoS measures. We inspected the block explorer of Algorand and found blocks with a number of transactions ranging from 1 to 7 (which was the largest block that we could find throughout the entire experiment). We then processed each block 1000 times and computed the average processing time. We then ran a regression on these 7 points to estimate the line

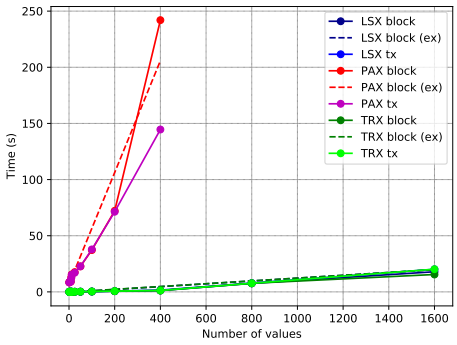
$$\gamma = 0.0006306x + 0.3003$$

which gives the average processing time γ as a function of the number number of transactions x.

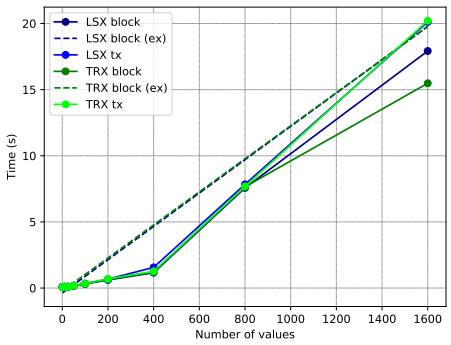
We then computed the number of blocks needed to store a fixed number of values and multiplied that by the average processing time to get an estimate. In Figure 5.6, we set x to 10 and show the simulated results in dashed lines. We can see that the slope of the dotted lines is more than the original line which indicates that the query time of the schemes (when using block numbers as addresses) depend on the block size.

5.8.2 Storage Complexity

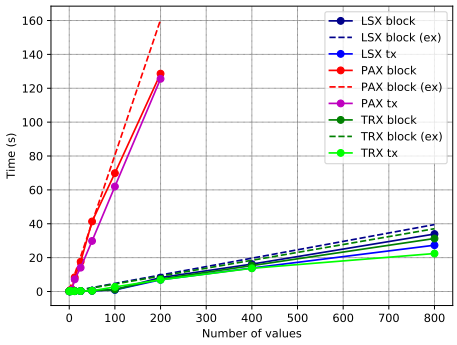
We now estimate the storage complexity of our schemes on Ethereum and Algorand. We compute the number of transactions needed and multiply that with the space taken by each transaction. We estimate the storage cost under three scenarios: (1) before any deletes; (2) after sparse deletes; and (3) after dense deletes. We further subdivide each scenario into two: (1) separate-updates; and (2) bulk-updates. In the former, we assume all the values are added (deleted) using separate Edit^+ (Edit^-) calls, and in the latter we assume all are added (deleted) in a single Edit^+ (Edit^-) call. The former measures the maximum space overhead while the latter measures the minimum space



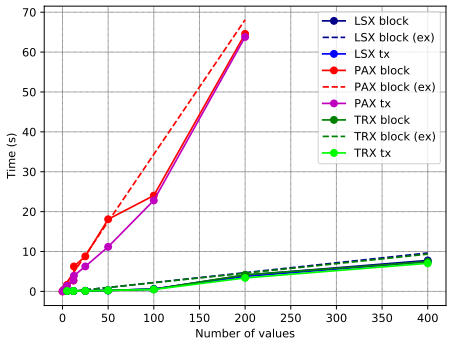
(a) Query before deletes



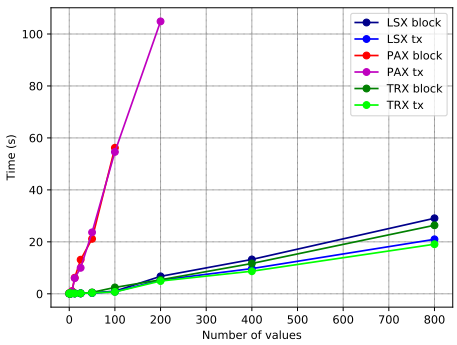
(b) Query before deletes (omitting PAX)



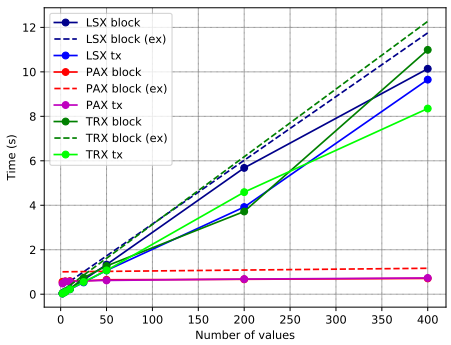
(c) Query after sparse deletes



(d) Query after dense deletes



(e) Query after random deletes



(f) Query after all-but-one deletes

Figure 5.6: Query complexity for Algorand before and after different forms of deletes. The dashed lines plot the simulated data assuming blocks contain 10 transactions each.

overhead. We detail our results in Tables 5.2 and 5.3 to store 1MB and 100MB of data on Ethereum and Algorand blockchains respectively.

Notation. Let V be the initial number of label/value pairs in the multi-map. For simplicity, we assume each value is 1B long.[5]. Therefore, in total, the size of the multi-map is V bytes. Further let f be the size of the fixed fields in a transaction and λ be the maximum size of the variable-sized data field in a transaction (both in bytes).

LSX. Before any deletions and if all values are added in a single transaction, LSX takes at least

$$\left\lceil \frac{V}{\lambda - r} \right\rceil \cdot \left(f + \lambda \right)$$

bytes to store a V-byte multi-map. This is because the number of transactions needed to store V bytes is at least $\lceil V/(\lambda - r) \rceil$ and each transaction takes $(f + \lambda)$ bytes of storage. If the values are added separately (i.e., in different transactions), LSX creates V separate transactions, where each transaction store an r-byte long pointer to the previous transaction and 1B long value. LSX, therefore, takes at most $V(f + r + 1)$ bytes to store a V-byte multi-map.

Since LSX handles deletions through additions, deleting a value is equivalent to adding a value. Therefore, deleting $V/2$ values is equivalent to adding $V/2$ values to the already existing V values. Moreover, since LSX treats both sparse and dense deletes similarly, the min and max values are the same. More precisely, LSX takes in total at least

$$\left\lceil \frac{3V}{2(\lambda - r)} \right\rceil \cdot \left(f + \lambda \right)$$

bytes and at most $3V(f + r + 1)/2$ bytes after both sparse and dense deletes.

TRX. The analysis is exactly the same as for LSX with the difference that we store two addresses in each transaction instead of one and therefore all r's in the expressions are replaced with $2r$.

PAX. Before any deletions, PAX takes $V(f + r + 1)$ bytes to store a V-byte multi-map. This is because PAX stores all the values in separate transactions with each transaction storing a single value and a single address. There is also no patch structure at this point.

Recall that a patch contains a left pointer, a right pointer and data of the form $(s \to p) \| \mathsf{lp} \| \mathsf{rp}$. All four quantities are addresses so patch is $4r$ bytes long. After sparse deletes, PAX requires

$$\frac{V}{2}(f + r + 1) + \frac{V}{2}(f + 4r) = \frac{V}{2}(2f + 5r + 1)$$

bytes of space. This is because there are $V/2$ transactions with un-deleted values, each of which is $(f + r + 1)$ bytes long, and there are $V/2$ transactions for patches in the patch tree, each of which is $(f + 4r)$ bytes long. After dense deletes PAX takes

$$\frac{V}{2}(f + r + 1) + (f + 4r)$$

[5]The estimates can be trivially extended to use a different value size.

	Before deletes		After sparse deletes		After dense deletes	
	min	max	min	max	min	max
Eth: LSX	1.0	159.0	1.5	238.5	1.5	238.5
Eth: TRX	1.0	191.0	1.5	286.5	1.5	286.5
Eth: PAX	159.0	159.0	206.5	206.5	79.5	79.5
Algo: LSX	1.25	237.0	1.87	355.5	1.87	355.5
Algo: TRX	1.32	289.0	1.98	433.5	1.98	433.5
Algo: PAX	237.0	237.0	314.5	314.5	118.5	118.5

Table 5.2: Storage Estimates (in MBs) for Ethereum and Algorand blockchains for storing 1 MB of data. For Ethereum: $V = 1$ MB, $f = 126$ B, $\lambda = 98$ KBs, $r = 66$ B. For Algorand: $V = 1$ MB, $f = 184$ B, $\lambda = 1$ KB, $r = 52$ B.

bytes since the patch tree only contains a single patch. Since PAX does not pack multiple values in one transaction, its storage overhead is independent of whether updated are done individually or all in one transaction. Therefore, all the min and max values are the same for PAX.

Storage estimates for Ethereum and Algorand. We estimated the storage overhead for storing 1MB and 100MB of data on Ethereum and Algorand. That is, we set $V = 1$MB and $V = 100$MBs.

For Ethereum, we analysed the fields of its transactions and estimated that a transaction takes 126 bytes (excluding the variable data) so we $f = 126$ bytes. We also set $\lambda = 98$KB, which is the maximum amount of data that can be stored in an Ethereum transaction (since the current gas limit per Ethereum block is 4.7 million gas). Finally, we set $r = 32$ bytes, which is the size of a transaction hash. We did a similar analysis for Algorand and set $f = 184$ bytes, $\lambda = 1$ KB, and $r = 52$ bytes[6]. The storage overhead estimates are summarised in Tables 5.2 and 5.3.

As expected, in the best case LSX and TRX do much better than PAX. This is because, due to packing, they create a smaller number of transactions and can amortize some of the storage costs of transactions. Also, in the worst case, when clients make individual updates, LSX and TRX are unable to pack and hence perform approximately the same PAX. However, if deletes are dense, PAX maintains only a single patch in the patch tree, whereas both LSX and TRX maintain a long history of deleted values which makes their storage much larger.

[6]We point out that for Algorand the address is not really the hash but a transaction id which they compute by encoding the signed transaction in Base64.

	Before deletes		After sparse deletes		After dense deletes	
	min	max	min	max	min	max
Eth: LSX	0.1	15.9	0.15	23.85	0.15	23.85
Eth: TRX	0.1	19.1	0.15	28.65	0.15	28.65
Eth: PAX	15.9	15.9	20.65	20.65	7.95	7.95
Algo: LSX	0.12	23.7	0.19	35.55	0.19	35.55
Algo: TRX	0.13	28.9	0.2	43.35	0.2	43.35
Algo: PAX	23.7	23.7	31.45	31.45	11.85	11.85

Table 5.3: For Ethereum: $V = 1$ MB, $f = 126$ B, $\lambda = 98$ KBs, $r = 66$ B. For Algorand: $V = 1$ MB, $f = 184$ B, $\lambda = 1$ KB, $r = 52$ B.

Bibliography

[1] Algorand. https://algorand.com/.

[2] Apache ignite. https://ignite.apache.org/.

[3] Bittorrent. https://www.bittorrent.com/.

[4] Couchbase. https://www.couchbase.com/.

[5] Ethereum. https://ethereum.org/.

[6] Foundationdb. https://www.foundationdb.org/.

[7] Memcachedb. https://github.com/LMDB/memcachedb/.

[8] A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.

[9] Redis. https://redis.io/.

[10] Xap. https://www.gigaspaces.com/.

[11] The monero project, 2014. https://web.getmonero.org/.

[12] Wanchain, 2018. https://www.wanchain.org/.

[13] Algorand's algosdk package, 2020. https://github.com/algorand/py-algorand-sdk.

[14] Bluzelle, 2020. https://bluzelle.com/.

[15] Covenantsql, 2020. https://covenantsql.io/.

[16] Infura, 2020. https://infura.io/.

[17] Metamask, 2020. https://metamask.io/.

[18] Oursql, 2020. http://oursql.org/.

[19] Provendb, 2020. https://www.provendb.com/litepaper/.

[20] Purestake, 2020. https://www.purestake.com/.

[21] Web3 documentation, 2020. https://web3py.readthedocs.io/en/stable/web3.eth.html.

[22] Daniel Adkins, Archita Agarwal, Seny Kamara, and Tarik Moataz. Encrypted blockchain databases. Cryptology ePrint Archive, Report 2020/827, 2020. https://eprint.iacr.org/2020/827.pdf.

[23] A. Agarwal and S. Kamara. Encrypted distributed hash tables. Technical Report 2019/1126, IACR ePrint Cryptography Archive, 2019.

[24] Archita Agarwal and Seny Kamara. Encrypted distributed hash tables. Cryptology ePrint Archive, Report 2019/1126, 2019. https://eprint.iacr.org/2019/1126.

[25] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD International Conference on Management of Data*, pages 563–574, 2004.

[26] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. In *Proceedings on Privacy Enhancing Technologies (Po/PETS '19)*, 2019.

[27] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[28] G. Asharov, M. Naor, G. Segev, and I. Shahaf. Searchable symmetric encryption: Optimal locality in linear space via two-dimensional balanced allocations. In *ACM Symposium on Theory of Computing (STOC '16)*, STOC '16, pages 1101–1114, New York, NY, USA, 2016. ACM.

[29] Gilad Asharov, Gil Segev, and Ido Shahaf. Tight tradeoffs in searchable symmetric encryption. In *Annual International Cryptology Conference*, pages 407–436. Springer, 2018.

[30] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *2016 2nd International Conference on Open and Big Data (OBD)*, pages 25–30. IEEE, 2016.

[31] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772, 2013.

[32] Silvia Bartolucci, Pauline Bernat, and Daniel Joseph. Sharvot: secret share-based voting on the blockchain. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 30–34, 2018.

[33] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO '07*, Lecture Notes in Computer Science, pages 535–552. Springer, 2007.

[34] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[35] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a blockchain keep a secret? *IACR Cryptol. ePrint Arch.*, 2020:464, 2020.

[36] Christian Berger, Birgit Penzenstadler, and Olaf Drögehorn. On using blockchains for safety-critical systems. In *Proceedings of the 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 30–36, 2018.

[37] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *Network and Distributed System Security Symposium (NDSS '20)*, 2020.

[38] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill. Order-preserving symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, 2009.

[39] R. Bost. Sophos - forward secure searchable encryption. In *ACM Conference on Computer and Communications Security (CCS '16)*, 20016.

[40] R. Bost, B. Minaud, and O. Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *ACM Conference on Computer and Communications Security (CCS '17)*, 2017.

[41] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[42] John W Byers, Jeffrey Considine, and Michael Mitzenmacher. Geometric generalizations of the power of two choices. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 54–63. ACM, 2004.

[43] Chengjun Cai, Xingliang Yuan, and Cong Wang. Hardening distributed and encrypted keyword search via blockchain. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 119–128. IEEE, 2017.

[44] Chengjun Cai, Xingliang Yuan, and Cong Wang. Towards trustworthy and private keyword search in encrypted decentralized storage. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[45] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1), 2000.

[46] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[47] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *ACM Conference on Communications and Computer Security (CCS '15)*, pages 668–679. ACM, 2015.

[48] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[49] D. Cash and S. Tessaro. The locality of searchable symmetric encryption. In *Advances in Cryptology - EUROCRYPT 2014*, 2014.

[50] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel Rosu, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[51] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[52] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[53] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[54] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 202–215. ACM, 2001.

[55] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[56] I. Demertzis and C. Papamanthou. Fast searchable encryption with tunable locality. In *ACM International Conference on Management of Data (SIGMOD '17)*, SIGMOD '17, pages 1053–1067, New York, NY, USA, 2017. ACM.

[57] Ioannis Demertzis, Dimitrios Papadopoulos, and Charalampos Papamanthou. Searchable encryption with optimal locality: Achieving sublogarithmic read efficiency. In *Advances in Cryptology - CRYPTO '18*, pages 371–406. Springer, 2018.

[58] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 75–80. IEEE, 2001.

[59] Muhammad El-Hindi, Martin Heyden, Carsten Binnig, Ravi Ramamurthy, Arvind Arasu, and Donald Kossmann. Blockchaindb-towards a shared database on blockchains. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1905–1908, 2019.

[60] Mohammad Etemad, Alptekin Küpccü, Charalampos Papamanthou, and David Evans. Efficient dynamic searchable encryption with forward privacy. *PoPETs*, 2018(1):5–20, 2018.

[61] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich queries on encrypted data: Beyond exact matches. In *European Symposium on Research in Computer Security (ESORICS '15). Lecture Notes in Computer Science*, volume 9327, pages 123–145, 2015.

[62] B. A Fisch, B. Vo, F. Krell, A. Kumarasubramanian, V. Kolesnikov, T. Malkin, and S. M. Bellovin. Malicious-client security in blind seer: a scalable private dbms. In *IEEE Symposium on Security and Privacy*, pages 395–410. IEEE, 2015.

[63] Michael J Freedman, Eric Freudenthal, and David Mazieres. Democratizing content publication with coral. In *NSDI*, volume 4, pages 18–18, 2004.

[64] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC '09)*, pages 169–178. ACM Press, 2009.

[65] E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.

[66] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[67] Sven Helmer, Matteo Roggia, Nabil El Ioini, and Claus Pahl. Ethernitydb–integrating database functionality into a blockchain. In *European Conference on Advances in Databases and Information Systems*, pages 37–44. Springer, 2018.

[68] Shengshan Hu, Chengjun Cai, Qian Wang, Cong Wang, Xiangyang Luo, and Kui Ren. Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 792–800. IEEE, 2018.

[69] M. Saiful Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Network and Distributed System Security Symposium (NDSS '12)*, 2012.

[70] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 98–107. Springer, 2003.

[71] S. Kamara and T. Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *Advances in Cryptology - EUROCRYPT '17*, 2017.

[72] S. Kamara and T. Moataz. SQL on Structurally-Encrypted Data. In *Asiacrypt*, 2018.

[73] S. Kamara and T. Moataz. Computationally volume-hiding structured encryption. In *Advances in Cryptology - Eurocrypt' 19*, 2019.

[74] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[75] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[76] Seny Kamara, Tarik Moataz, and Olya Ohrimenko. Structured encryption and leakae suppression. In *Advances in Cryptology - CRYPTO '18*, 2018.

[77] Seny Kamara, Tarik Moataz, Stan Zdonik, and Zheguang Zhao. Opx: An optimal relational database encryption scheme. Technical report, IACR ePrint Cryptography Archive, 2020.

[78] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[79] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[80] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[81] Protocol Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf.

[82] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[83] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[84] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 313–328, 2013.

[85] Ricardo Macedo, João Paulo, Rogério Pontes, Bernardo Portela, Tiago Oliveira, Miguel Matos, and Rui Oliveira. A practical framework for privacy-preserving nosql databases. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 11–20. IEEE, 2017.

[86] Will Martino. Kadena: The first scalable, high performance private blockchain. *Kadena, Okinawa, Japan, Tech. Rep*, 2016.

[87] Gregory Maxwell and Andrew Poelstra. Borromean ring signatures. *Accessed: Jun*, 8:2019, 2015.

[88] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[89] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[90] X. Meng, S. Kamara, K. Nissim, and G. Kollios. Grecs: Graph encryption for approximate shortest distance queries. In *ACM Conference on Computer and Communications Security (CCS 15)*, 2015.

[91] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.

[92] Athicha Muthitacharoen, Robert Morris, Thomer M Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[93] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[94] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. *Proc. VLDB Endow.*, 12(11):1539–1552, 2019.

[95] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: design and implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, 2019.

[96] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and Communication Networks*, 9(18):5943–5964, 2016.

[97] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. Towards a novel privacy-preserving access control model based on blockchain technology in iot. In *Europe and*

MENA Cooperation Advances in Information and Communication Technologies, pages 523–533. Springer, 2017.

[98] Jeffrey Pang, Phillip B Gibbons, Michael Kaminsky, Srinivasan Seshan, and Haifeng Yu. Defragmenting dht-based distributed file systems. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 14–14. IEEE, 2007.

[99] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S.-G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 359–374. IEEE, 2014.

[100] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *Proceedings of the VLDB Endowment*, 12(11):1664–1678, 2019.

[101] Bruno Produit. Using blockchain technology in distributed storage systems. 2018.

[102] Mayank Raikwar, Danilo Gligoroski, and Katina Kralevska. Sok of used cryptography in blockchain. *IEEE Access*, 7:148550–148575, 2019.

[103] Jens-Andreas Hanssen Rensaa, Danilo Gligoroski, Katina Kralevska, Anton Hasselgren, and Arild Faxvaag. Verifymed–a blockchain platform for transparent trust in virtualized healthcare: Proof-of-concept. *arXiv preprint arXiv:2005.08804*, 2020.

[104] Manuj Subhankar Sahoo and Pallav Kumar Baruah. Hbasechaindb–a scalable blockchain framework on hadoop ecosystem. In *Asian Conference on Supercomputing Frontiers*, pages 18–29. Springer, 2018.

[105] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[106] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730. ACM, 2012.

[107] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[108] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable encryption with small leakage. In *Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[109] Moritz Steiner, Damiano Carra, and Ernst W Biersack. Faster content access in kad. In *Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on*, pages 195–204. IEEE, 2008.

[110] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[111] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[112] Andrew Tam. Secret with enigma: A walkthrough., 2018. https://blog.enigma.co/secret-voting-smart-contracts-with-enigma-a-walkthrough-5bb976164753.

[113] Basho Technologies. Riak. https://docs.basho.com/riak/kv/2.2.2/learn/dynamo/.

[114] Viktor Trón, Aron Fischer, Dániel A Nagy, Zsolt Felföldi, and Nick Johnson. Swap, swear, and swindle: Incentive system for swarm. *Technical Report, Ethersphere Orange Papers 1*, 2016.

[115] Sarah Underwood. Blockchain beyond bitcoin, 2016.

[116] Licheng Wang, Xiaoying Shen, Jing Li, Jun Shao, and Yixian Yang. Cryptographic primitives in blockchains. *Journal of Network and Computer Applications*, 127:43–58, 2019.

[117] Xiaoming Wang and Dmitri Loguinov. Load-balancing performance of consistent hashing: asymptotic analysis of random node join. *IEEE/ACM Transactions on Networking (TON)*, 15(4):892–905, 2007.

[118] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[119] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308, 2013.

[120] Yinghui Zhang, Robert H Deng, Jiangang Shu, Kan Yang, and Dong Zheng. Tkse: Trustworthy keyword search over encrypted data with two-side verifiability via blockchain. *IEEE Access*, 6:31077–31087, 2018.

[121] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. Minicrypt: Reconciling encryption and compression for big data stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 191–204, 2017.

[122] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471*, 2015.