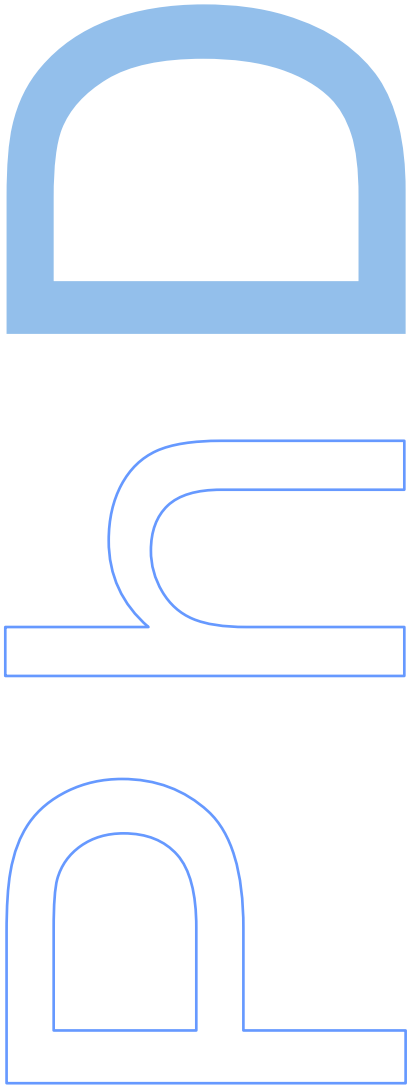
Trade-offs between privacy and efficiency on databases

Rogério Pontes
MAPi
DCC
2021

Orientador
Manuel Barbosa, Professor Associado do Departamento de Ciência de Computadores, Faculdade de Ciências da Universidade do Porto

Coorientador
Ricardo Vilaça, Investigador Auxiliar do Departamento de Informática, Escola de Engenharia da Universidade do Minho



Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: Rogério Pontes DATE: 29/01/2021

Aos meus pais, António Silva e Maria Silva.

Agradecimentos

Só é possível chegar a bom porto na viagem infindável da descoberta de conhecimento com apoio, orientação e companheirismo. Estou imensamente grato a todos os que me ajudaram a concluir mais uma fase fundamental desta viagem.

Uma parte essencial do doutoramento é a discussão de ideias que, neste caso, se tornou interessante e estimulante devido ao conhecimento, criatividade e acima de tudo paciência dos meus orientadores. Quero agradecer ao professor Manuel Barbosa pela sua disponibilidade, interesse e extrema dedicação ao debate de ideias até chegar ao cerne do problema. Esta determinação transformou-se em inspiração e modelo a seguir na abordagem dos problemas que surgem em investigação. Estou também grato ao Doutor Ricardo Vilaça pelo seu apoio incondicional que sempre me incentivou a enfrentar novos desafios e a superar os obstáculos que inevitavelmente surgem.

Estou muito grato por ao longo deste percurso ter podido sempre contar com os docentes do grupo de investigação do HASLab que me proporcionaram um ambiente de trabalho fascinante onde se preza a colaboração e interajuda. Certamente que sem o apoio prestado pelo Prof. José Nuno e do Prof. Rui Oliveira em me integrar no grupo que não teria percorrido este caminho. Agradeço também ao Prof. José Pereira que esteve sempre disposto para discutir o meu trabalho.

Foi uma experiência incrível passar estes anos da minha vida a trabalhar, mas acima de tudo conviver e aprender com todos os colegas que estão ou passaram pelo laboratório. A vossa energia e união torna qualquer momento difícil melhor e todos os bons momentos excecionais. Estou profundamente grato pelo vosso apoio e ajuda, Ana Nunes, Bernardo Portela, Cláudia Brito, Daniel Cruz, Diogo Couto, Fábio Coelho, Francisco Cruz, Francisco Maia, Francisco Neves, Georges Younes, João Paulo, Miguel Matos, Nuno Machado, Paula Rodrigues, Ricardo Gonçalves, Ricardo Macedo, Rui Ribeiro, Tânia Gomes, Tiago Oliveira e Vítor Duarte.

I am also grateful for the opportunity to collaborate and learn with Professor Pascal Felber, Dr. Huges Mercier, Dr. Valerio Schiavoni and Dorian Burihabwa from the university of Neuchâtel, Switzerland.

Não existem palavras que descrevam a gratidão que tenho pela minha família. É imensurável

o apoio e ajuda dos meus pais, António Silva e Maria Silva. Sem o vosso exemplo de empenho e sacrifício nunca teria conseguido alcançar este objetivo. Tenho uma enorme dívida para com os meus irmãos, Bruno Pontes e Sónia Pontes, que sempre lutaram as batalhas que eu não pude lutar.

Em todas as longas viagens existem momentos onde o futuro é incerto e parece que desvanece. Estou eternamente grato à Vera Matos por me ter acompanhado e encorajado a percorrer o meu caminho, mesmo quando isso significava não podermos criar recordações todos os dias.

A minha gratidão vai também para as instituições que apoiaram o trabalho desta tese. A Fundação para Ciência e Tecnologia (FCT) apoio este trabalho através da bolsa de doutoramento (SFRH/BD/142704/2018). O Departamento de informática da Universidade do Minho, o HASLab - High Assurance Software Lab, o INESC TEC e ao departamento de ciência de computadores da Universidade do Porto.

FCT Fundação para a Ciência e a Tecnologia

Abstract

Outsourcing data to third-party services, most commonly public cloud providers, has become a default practice by individuals and organizations alike. By using a cloud environment, users can access their data at any time and from anywhere without having to invest in a private infrastructure. However, as soon as data is inside a cloud provider the user loses all control. The user becomes powerless to decide which security mechanics are used to prevent data from being disclosed. Cryptographically protected databases address this issue by encrypting data before it leaves the user's control. Nonetheless, classical symmetric schemes cannot be used without limiting the database functionality and performance. To circumvent this limitation, existing systems leak some partial information to process simple key-value queries as well rich relational queries over ciphertext. But this approach is exploitable as the information disclosed is susceptible to statistical attacks that can be used to reconstruct sensitive plaintext values.

In this dissertation we focus on the challenges of storing sensitive data on untrusted third-parties, processing database queries over encrypted data and exploring novel trade-offs between privacy, performance and security. Within the spectrum of possible trade-offs our goal is to propose novel systems that minimize the information disclosed and maximize the computation outsourced. Our first contribution, SafeFS, is a new multi-layer user-space encrypted file system. This solution stores the contents of a database system to a third-party service by intercepting file system requests and processing data in a pipeline of configurable layers that can encrypt, replicate and compress information. The following contribution, d'Artagnan, outsources data as well as the database engine by decentralizing information across multiple cloud providers. With this contribution, a single cloud breach and data disclosure does not impact the systems security. Furthermore, this system can process any database query and scale horizontally to leverage the cloud resources. Our third and final contribution, CODBS, presents a novel oblivious index scan scheme to search optimized index data structures on relational databases. This construction hides the access patterns disclosed by databases searches, preventing the adversary from learning when database records are accessed, how many times a database record is accessed and the size of a query result set. This construction has twice the throughput of state-of-the-art constructions. All of our contributions are experimentally evaluated with standard database benchmarks.

Resumo

A metodologia atual das organizações que procuram tornar a sua informação constantemente acessível é delegar o armazenamento e processamento para infraestruturas externas. Este processo de externalização é tipicamente feito para ambientes na nuvem. Todos os dados armazenados na nuvem são guardados em bases de dados que carecem de mecanismos de segurança. Para além disso, assim que os dados se encontram na nuvem, o utilizador deixa de ter controlo sobre os mesmos. Os dados podem ser copiados, divulgados ou propositadamente extraviados por um agente malicioso. Para mitigar este problema foram propostos sistemas de bases de dados criptográficos que cifram os dados antes de serem transmitidos para a nuvem. Contudo, os esquemas clássicos de cifras simétricas não podem ser utilizados sem limitar a funcionalidade e desempenho das bases de dados. Atualmente, a solução mais utilizada para quebrar esta tensão entre desempenho, segurança e funcionalidade é libertar propositadamente alguma informação parcial sobre os dados. Porem, a informação libertada é suscetível a ataques de inferência estáticos capazes de reconstruir os dados originais.

Esta dissertação debruça-se sobre os desafios de processar questões sobre texto cifrado e explora novos compromissos entre funcionalidade, segurança e desempenho. Dentro do espectro possível de soluções, o nosso objetivo é propor novos sistemas que minimizam a informação revelada durante o processamento de questões e maximizam a computação delegada para a nuvem. A nossa primeira contribuição é SafeFS, um sistema de ficheiros com múltiplas camadas dentro do espaço do usuário. Este sistema intercepta os pedidos ao sistema de ficheiro e, dependendo da configuração, pode replicar, cifrar ou comprimir os dados antes de serem reencaminhados para a nuvem. A segunda contribuição delega toda a computação e armazenamento para serviços de terceiros. Nesta contribuição propomos d'Artagnan, um sistema de base de dados NoSQL descentralizado que reparte os dados sensíveis em vários segredos. Os segredos são dispersados por várias nuvens independentes, assegurando que uma violação de segurança numa das infraestruturas não compromete a privacidade de todo o sistema. Na última contribuição apresentamos um novo esquema criptográfico que permite fazer pesquisas em base de dados relacionais sem revelar os padrões de acesso do sistema. Mais concretamente a construção proposta protege os registos acedidos, a ordem e número de acessos. O desempenho assimptótico e prático desta construção é duas vezes superior ao estado da arte. Todos os sistemas propostos foram validados experimentalmente.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CI Confidence Interval. 131

CODBS Cascading Oblivious Database Search. 136, 140

CPD Cryptographically Protected Databases. 14, 15, 19, 21, 37, 39, 49, 50, 77, 98, 138, 140

DET Deterministic Encryption Scheme. 24, 27, 47

FDW Foreign Data Wrapper. 130

Fuse Filesystems in Userpace. 42, 43, 50, 54, 56–61, 64–66, 74

IEE Isolated Execution Environment. 34–36, 42, 46, 47, 49, 50, 99, 121, 136, 140

IND-CPA Indistinguishability under chosen-plaintext attack. 22, 23, 25, 26, 31, 49, 104, 111, 122, 124, 127

IND-DCPA Indistinguishability under distinct chosen-plaintext attack. 25, 26

NFS Network File System. 42, 43

OIS Oblivious Index Scan. 106–108

OPE Order Preserving Encryption. 26, 27, 47, 50

ORAM Oblivious Random Access Machine. 16, 99, 100, 102, 103, 107, 109–111, 113–115, 121, 122, 125, 126, 128, 129, 136, 139, 140

POPF-CCA Pseudorandom order-preserving function against chosen-ciphertext attack. 26

PPE Property Preserving Encryption. 47, 48, 50, 77, 78

PPT Probabilistic Polynomial Time. 22–25, 27–30, 33–35, 37

PRF Pseudorandom Function. 23, 26, 48, 100, 104, 107, 110, 111, 113, 122, 124, 126, 130, 133

RAM Random Access Machine. 35

RDBS Relational Database Management Systems. 46

SDS Software-Defined Storage. 54

SMPC Secure multiparty computation. 16, 32, 33, 40, 78, 81, 82, 84, 90, 91, 94–96

SOC Secure Outsourced Computation. 5, 35–37

SSE Symmetric searchable encryption. 5, 27–29, 44–48, 50, 98–100, 102, 140

YCSB Yahoo! Cloud Serving Benchmark. 5, 90–95, 101, 131, 133, 135

Chapter 1

Introduction

Global production of data and consumption of information has far exceeded the computational resources of endpoint devices (e.g.: smart-phones, tables and laptops). Data is generated by a multitude of different sources, including personal interactions and professional transactions which have been digitalized and enhanced through email, messaging services and video conferencing. This digitalization process has extended to almost every major human activity, from production of resources, to retail and even the financial sector. It is expected that at 2025, 75 % of the global population will interact with data every 18 seconds. Each of these interactions is likely to increase the total amount of data available to 90ZB [RGR18]. The overlap of the real-world with the digital world created the expectation of immediate and on-demand access to online products and goods. This expectation, shared by the majority of users of online services, leads to an increasingly interconnected world where messages are constantly transmitted. However, this frenetic exchange of information cannot be sustained only by private consumer devices even despite the most recent technological advancements.

To handle the ever-increasing demand for data and information online services rely on cloud providers. In the cloud paradigm, physical computational resources such as data storage and computation are made available as a utility through an abstract concept of virtual resources. Individuals and organizations alike can allocate these resources at will without having to invest a large capital to acquire hardware and software infrastructure. Additionally, there is no need to pre-emptively plan how many resources need to be allocated as they can be seamlessly scaled. Public clouds providers such as Amazon, Google and Microsoft have allured a large number of clients by providing affordable storage and application hosting as well cloud-based products with high-availability and high-performance [AFG+09].

One salient issue of the cloud paradigm is the implicit assumption that users have to relinquish control over their data and entrust their privacy to the provider [CGJ+09]. As soon as data is stored on the cloud there is no guarantee that the provider will not do an unsolicited back-up, analyse the data or remove it on requested. This concern is further aggravated when the

end-user of online services is unaware that their data is controlled by multiple entities that can be compromised. In fact, several reports exposed malicious system administrators that use their elevated privileges within an organization to disclose large quantities of data such as financial or medical records [FW19]. Even simple mistakes as invalid configurations or displacement of credential management can result in leaks of private sensitive information [Bri18, Gar19]. Besides the threat of internal administrators there are always malicious external attackers that seek to exploit any security vulnerability in the cloud infrastructure to gain privileged access and steal valuable information such as user accounts, passwords, emails and credit cards [Hau19]. Governments and national intelligence agencies are also interested in obtaining confidential information and legally force technological companies to either explicitly disclose the user's information or install backdoors in their services [Gre14].

To protect the end user's privacy, data must be encrypted at all times. New regulations have been issued to shift control away from online services and back to the end users. A prime example is the General Data Protection Regulation in the European Union which mandates applications to explicitly state all of data they collect, how this data is used and stored [EUd18]. These regulations further incentivize applications to encrypt data as much as possible by lowering the accountability of a company if only encrypted data is leaked. While there are efficient cryptographic protocols that protect the user's confidentiality by encrypting data in transit, when messages are exchanged between users and applications, and at rest, when data is persisted in long term storage, the same does not hold when data has to be processed. As such, online services resort to database management systems with minimal privacy mechanisms that store and process unencrypted data [FVY+17], often hosted in public cloud environments.

Classical database systems are optimized to search data efficiently and are not suited to process encrypted data. The main goal of these systems is to provide a general purpose datastore that processes queries efficiently [Sto10]. Data is always handled as plaintext and the only existing security safeguards in place are identity and access management mechanisms. However, these safeguards are not sufficient to protect the user's confidentiality as any cloud administrator can simply access the database storage directly. Even a single data dump could be useful to malicious external attackers. Ideally, the data stored and used in a database should be encrypted to provide better security guarantees. One possible naive approach is to encrypt the database as a single file on the client side before storing it on a third-party service. This approach does not require any modification to the database engine and the client only has to manage its own private encryption keys. But clearly this solution limits the databases ability to process queries. Database queries cannot be processed on the server side as the database engine is not capable of inferring any information from the encrypted data. To process a query the client would have to download the entire database, which is in most cases prohibitively slow, i.e., any query is evaluated in linear time to the database size. Additionally, client's endpoint devices have limited computational power and storage capacity.

More refined solutions have proposed to use searchable encryption schemes [BHJP14]. These schemes store data in index data structures that map encrypted keywords to documents. The documents as well as queries are encrypted on a trusted site, the client, before being uploaded to an untrusted third-party. Using cryptographic tokens generated by the client, the server can search the index for a set of matching records and output the set of correct results. The results can only be decrypted by the client which keeps a small state containing a secret key. One appealing feature of searchable encryption schemes is the provable security approach used to formally define the security guarantees, the trust model where the guarantees hold and even what a malicious attacker is allowed to do and learn. However, any operation in a searchable encryption index has an associated leakage that enables the server to evaluate queries. This leakage may for instance disclose the number of results and the records that satisfy a query, but never the plaintext value of a record. However, the information disclosed is still useful for a malicious attacker to learn sensitive information. While these schemes can search encrypted data in sub-linear time, the set of queries supported is restricted to simple key-based searches.

To expand the set of queries supported, Cryptographically Protected Databases (CPD) have been proposed as a more general solution. Instead of using only searchable encryption schemes, these systems use a specific scheme for each database operator [FVY+17]. To sort or join data, property-preserving schemes such as order-preserving encryption can be used. These encryptions schemes generate ciphertexts that maintain the total order of the plaintexts and enable the database to compare values without decrypting them. Arithmetic operations, such as multiplication and addition, can also be calculated over ciphertext by using partially homomorphic encryption schemes. Similar to searchable encryption schemes, data is encrypted on the client side before it is stored on the server and queries are also evaluated without any client-side help. The main problem of this approach is the combination of multiple encryption schemes often without considering their security models which enables malicious attackers to gain access to several sources of information leakage. The combination of all of the leakages results in disclosing more information than any individual protocol [NKW15].

1.1 Problem statement and objectives

New CPD systems need to supersede classical databases that lack the security mechanisms to protect its data confidentiality in increasingly complex technological environments. Existing solutions to address this security gap fall within a spectrum of possible approaches. The spectrum ranges from systems that prioritize security over performance to systems that value practical application and performance over security guarantees. At the security extreme of the spectrum there are homomorphic encryption schemes capable of evaluating any arbitrary function over encrypted text and ensure semantic security, i.e., an untrusted third-party does not learn any information about plaintexts from encrypted data besides its size. However, existing

constructions have a significant performance overhead and are not suitable for production databases. As we move away from this extreme, we find cryptographic schemes with controlled leakage that disclose partial information to obtain increasingly efficient solutions. At the middle of the spectrum there are searchable encryption schemes that support a subset of database operators with limited performance overhead and data leakage. Individually, a cryptographic scheme is not sufficient to satisfy a full fledge database requirements, either due to performance or functional limitations. By moving further into the opposite extreme, we have CPDs that use multiple cryptographic schemes, often irreconcilable, to support a complete set of database operations. At this extreme, there are systems such as CryptDB [PRZB11] that are capable of supporting several relational queries with minimal performance overhead. However, these solutions reveal some information about the encrypted data such as the equality, order and range of values. As such, these systems should not be applied as general solutions and must carefully consider the security model assumed. Even though leaking some information is unavoidable, we aim to address the following research question in this dissertation:

> Can we identify new practically-relevant intermediate trade-offs between security and performance?

Our main goal in this dissertation is to explore novel trade-offs between performance and privacy in the spectrum of cryptographic protected databases. We split these goal in in tree parts. First, our objective is to design a modular encrypted file system that uses interchangeable security layers. This solution is intended for the most risk adverse applications that need to analyse data with expressive queries (e.g.: medical or financial applications). With this solution, queries should be processed on the client-side, but the data should be stored remotely on a third-party without requiring any database modification or downloading the entire database. Our second objective is to create a secure database system that decentralizes confidential data by dividing it in multiple secrets stored across multiple independent cloud providers. This approach should process queries with minimal client-side processing and prevent a security exploit in a single provider from compromising the entire system. The third and final objective is to take a more intrusive, bottom-up approach to relational database systems to construct an optimized secure index data structure. In this last objective we seek to optimize data structures in classical databases while leaking minimal information.

1.2 Contributions

Our contributions can be framed in the general problem of secure computation and storage outsourcing. In this setting, a client that wants to offload storage as well as computation to an untrusted site. The client resides on a trusted site that is assumed to be secure but has limited storage and computational resources. In contrast the untrusted site has virtually unlimited

resources but is susceptible to malicious attacks. Considering this setting, the contributions of this dissertation are the following.

Our first contribution keeps the computation on the client-side and is an encrypted remote file system that is placed as a middleware between the database engine and the untrusted party. A few similar solutions have been proposed in the literature, but existing systems are monolithic and support a limited set of cryptographic schemes. We designed and implemented SafeFS, a multi-layer encrypted file system optimized for private databases. Layers in SafeFS are configurable and compose to provide the trade-off between security and performance fine-tuned to an application requirements and workloads. The system has layers to encrypt, compress and replicate data through multiple third-party infrastructure. New layers can be integrated to extend the features of the file system through a plug-in system.

In our next contribution we propose a new system that moves some computation to the server side with a decentralized approach. Whereas cryptographically protected database systems rely on property-preserving schemes, we design a distributed database that leverages Secure multiparty computation (SMPC) [Gol04]. Our contribution is d'Artagnan, a trusted NoSQL database on untrusted clouds. In this solution, the untrusted site contains multiple independent clouds each storing only a part of the sensitive information. Contrary to property-preserving schemes, a single multiparty construction can be used to evaluate any query and the corruption of a subset of parties does not compromise the entire system. Furthermore, the trusted site does not need to process any part of the queries and only stores the credentials to the cloud providers. However, our current implementation is limited to NoSQL systems and the set of supported queries is restricted to key-value searches, including range queries.

Our third and final contribution tackles the challenge of supporting secure query processing on relational database. One of the critical sources of leakages in existing systems are the access patterns and the result size of queries evaluated on the server-side [IKK12]. To address these sources of leakage we propose CODBS, a cryptographic scheme that builds oblivious searches from the ground. Similar to searchable encryption schemes our scheme uses a secure index data structure. However, instead of integrating a new data structure as a black box in a relational database our construction is based on existing indexes optimized for relational queries. Our scheme addresses the existing sources of leakage by using Oblivious Random Access Machine (ORAM) schemes that hide the access patterns of remote accesses to an untrusted storage device. We additionally use trusted hardware that provides a secure execution environment that can evaluate our construction isolated from any external intrusion.

1.3 Results

The first two contributions of this dissertation, SafeFS and d'Artagnan, have been published as scientific papers in peer-reviewed conferences. Our last contribution, CODBS, is in the

submission process.

[PMPV16] SafeRegions: Performance Evaluation of Multi-party Protocols on HBase

R. Pontes, F. Maia, J. Paulo, R. Vilaça SRDS Workshop '16

This paper presents an initial prototype of d'Artagnan in the setting of NoSQL databases without any optimizations and support for only exact match queries. d'Artagnan extends upon this work by adding support for range queries, decoupling the secure multiparty protocols from the database architecture and with a system design that scales horizontally.

[PBM+17] SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All

R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, R. Oliveira SYSTOR '17

This paper presents SafeFS, the design principles behind the system and the its architecture. The main results of this publication are our encrypted file system and the validation of the multi-layer approach with a practical experimental evaluation. The evaluation shows that stacking multiple logical layers, to ensure privacy and provide high-availability, has a minimal performance overhead over native file systems. This work was developed in the context of SafeCloud [1], an European research project, and the resulting framework is public available [2].

[PMVM19] d'Artagnan: A Trusted NoSQL Database on Untrusted Clouds

R. Pontes, F. Maia, R. Vilaça, N. Machado SRDS '19

This paper presents d'Artagnan, the first privacy-aware multi-cloud NoSQL database framework. The main result of this publication is a framework that decentralizes sensitive data over multiple independent cloud providers, preventing a single breach from compromising the entire system. This framework evaluates queries with optimized secure multiparty protocols but is designed to support the integration of new protocols with different security models. The framework was experimentally validated in private clusters as well as in public clouds. This project was also developed in the context of the SafeCloud project and is also publicly available [3].

Building oblivious search from the ground up

[1] https://www.safecloud-project.eu
[2] https://github.com/safecloud-project/safefs
[3] https://dbr-haslab.github.io/tools/dartagnan/

R. Pontes, M. Barbosa, B. Portela, R. Vilaça (In Submission)

In this paper the authors propose CODBS, an oblivious search protocol for relational databases. The main result of this paper is an ORAM-based cryptographic construction that replaces classical index data structures used in relational databases. The resulting scheme can search the database efficiently with minimal information leakage and has a better asymptotic performance than state-of-the-art solutions. This performance improvement is experimentally validated with the integration of the protocol on PostgreSQL, one of the most widely used open-source databases. The results of this contributions are public available [4].

Beside the core results of this dissertation, the author collaborated on additional novel research on storage and database security. These collaborations resulted in conference publications that are not included in the main contributions of the dissertation but were nonetheless instrumental in defining our research approach. Some of these publications report preliminary work while others explore trade-offs in different database systems.

[BPF+16] On the Cost of Safe Storage for Public Clouds: An Experimental Evaluation

D. Burihabwa, R. Pontes, P. Felber, F. Maia, H. Mercier, R. Oliveira, V. Schiavoni, J. Paulo SRDS '16

This paper presents a testbed and an extensive evaluation of cryptographic schemes that can be efficiently used to outsource sensitive data to the cloud. The evaluation considered a single-cloud and a multi-cloud setting. This work was a first step towards SafeFS.

[PPB+17] Performance trade-offs on a secure multi-party relational database

R. Pontes, M. Pinto, M. Barbosa, R. Vilaça, M. Matos, R. Oliveira SAC '17

In this paper, the idea of a multiparty database is explored in the setting of relational databases. The main contribution of this paper is a performance optimization on the protocols to lower the overall query bandwidth and improve query latency. This optimization is integrated on the d'Artagnan framework.

[MPP+17] A Practical Framework for Privacy-Preserving NoSQL Databases

R. Macedo, J. Paulo, R. Pontes, B. Portela, T. Oliveira, M. Matos, R. Oliveira SRDS '17

[4] https://github.com/rogerioacp/SOE

This paper proposes a modular and extensible encrypted NoSQL data store that can use multiple cryptographic schemes to store sensitive data and process queries. Whereas the majority of existing research was focused on relational databases, this paper addresses the challenges of designing a highly-scalable, high-performance cryptographic protected NoSQL database.

[CCP+20] On the trade-offs of combining multiple secure processing primitives for data analytics

H. Carvalho, D. Cruz, R. Pontes, J. Paulo DAIS '20

This paper presents an initial prototype of a privacy-aware data analytical engine. This prototype is used to quantitatively measure in an experimental evaluation which cryptographic schemes, including trusted hardware, provide the best trade-offs between performance and security in a data analytics setting that process mostly arithmetic and aggregation queries.

1.4 Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 defines the correctness and security model of the fundamental cryptographic schemes relevant to this dissertation.
- Chapter 3 surveys the related work on encrypted file systems as well as CPD.
- Chapter 4 presents the first contribution of this dissertation SafeFS, a multi-layer file system for CPD.
- Chapter 5 presents the design, implementation and system evaluation of d'Artagnan, the second contribution of this dissertation on cryptographic protected databases.
- Chapter 6 presents CODBS, our third contribution, a novel cryptographic scheme for oblivious searches as well as its practical evaluation.
- Chapter 7 concludes this dissertation by discussing the main contributions and presenting a few interesting open research paths.

Chapter 2

Preliminaries

The security of CPD is based on a set of fundamental cryptographic primitives. These building blocks are independent cryptographic systems with precise security grantees proven within a trust model that requires a thorough understanding to ensure their correct application. This chapter provides an overview of the main primitives crucial to this thesis and the related work.

This chapter has the following structure: Section 2.1 provides the classical definition of symmetric encryption and pseudo-random functions; Section 2.2 presents security definitions of encryption schemes that can evaluate a limited set of functions over encrypted data by disclosing some partial information; Section 2.3 describes cryptographic mechanisms capable of evaluating any function over encrypted data, each one with a distinct execution and trust model.

2.1 Classic cryptographic primitives

2.1.1 Symmetric encryption

Symmetric encryption is a cryptographic scheme that provides confidentiality to two independent parties in a symmetric setting. In this setting, parties are interconnected over an insecure channel controlled by an eavesdropper. Parties can send and receive messages through the channel but want to ensure the messages remain confidential and the eavesdropper does not learn the messages contents. This can be accomplished with a symmetric encryption scheme, assuming that both parties share a secret key sk that is never disclosed to the eavesdropper. How the parties agree upon a common secret key is outside of the scope of the scheme. To exchange message securely, a party uses a symmetric encryption scheme to encrypt a plaintext message m and obtain a resulting ciphertext c. The ciphertext is sent over the network and the receiving party executes the inverse process to decrypt c and obtain the original plaintext message m.

Experiment $\text{IND-CPA}^0_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Experiment $\text{IND-CPA}^1_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Oracle $\text{Encrypt}_\text{b}(msg_0, msg_1)$
$\text{sk} \leftarrow\!\!\$\ \text{KGen}(1^\lambda)$	$\text{sk} \leftarrow\!\!\$\ \text{KGen}(1^\lambda)$	**if** $\lvert msg_0 \rvert \neq \lvert msg_1 \rvert$ **do**
return $\mathcal{A}^{Encrypt_0}(1^\lambda)$	**return** $\mathcal{A}^{Encrypt_1}(1^\lambda)$	**return** $\perp$
		return $\text{Enc}(\text{sk}, msg_b)$

Figure 2.1: Game definition of symmetric encryption scheme security.

Definition 2.1.1 (Symmetric Encryption Scheme)**.** A symmetric encryption scheme $\mathcal{E} = (\text{KGen}, \text{Enc}, \text{Dec})$ is defined by the following three Probabilistic Polynomial Time (PPT) algorithms:

$\text{KGen}(1^\lambda) \rightarrow \text{sk}$**:** Probabilistic key generation algorithm that given as input the security parameter 1^λ, outputs a new secret key sk sampled from a distribution underlying the scheme.

$\text{Enc}(\text{sk}, m) \rightarrow c$**:** *Probabilistic* encryption algorithm that takes as input a secret key sk and a plaintext message $m \in \{0,1\}^*$. It outputs a ciphertext $c \in \{0,1\}^*$.

$\text{Dec}(\text{sk}, c) \rightarrow m$**:** Deterministic algorithm that takes as input a secret key sk and a ciphertext $c \in \{0,1\}^*$. The result of this algorithm is a decrypted plaintext message $m \in \{0,1\}^*$.

A symmetric encryption scheme must satisfy a correctness property. This property ensures that the decryption of a ciphertext with a legitimate secret key returns a valid plaintext message. A scheme is said to be *correct* if for every secret key $\text{sk} \leftarrow\!\!\$\ \text{KGen}(1^\lambda)$ sampled from the key generation algorithm and for every input message m, it holds that:

$$m = \text{Dec}(\text{sk}, \text{Enc}(\text{sk}, m))$$

The goal of a symmetric encryption scheme is to ensure message confidentiality. This security guarantee is captured by the security game Indistinguishability under chosen-plaintext attack (IND-CPA), defined in Figure 2.1. The IND-CPA game consists of an adversary that runs in one of two distinct experiments. The adversary $\mathcal{A}$ is an algorithm that does not have access to the experiment internal actions and can only interact with the experiment through an oracle Encrypt_b. The oracle is a block box function that given as input a tuple of plaintext messages (msg_0, msg_1), it chooses a single message to encrypt and outputs a ciphertext. In the experiment $\text{IND-CPA}^0_{\mathcal{E},\mathcal{A}}$, the oracle always encrypts the first message msg_0. In experiment $\text{IND-CPA}^1_{\mathcal{E},\mathcal{A}}$ the choice is reversed. The game is initialized by sampling a random bit $b \leftarrow\!\!\$\ \{0,1\}$ that determines the oracle choice without the adversary knowledge. Additionally, a secret key sk is sampled by the experiment and is made available to the oracle but it is never accessible to the adversary. After the initialization the choice of which message is encrypted never changes. The adversary proceeds by requesting a polynomial number of messages. After each message it can dynamically adjust its attack strategy based on the oracle output. In the end, the adversary

guesses which one of the experiments it was playing against by outputting a bit b'. The adversary wins the game if the guess is correct (b=b').

Definition 2.1.2 ($\mathrm{IND\text{-}CPA}$ security)**.** Let $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme and $\mathcal{A}$ a PPT adversary with access to oracle $\mathsf{Encrypt_b}$. The IND-CPA advantage of $\mathcal{A}$ is defined as follows:

$$\mathsf{Adv}^{\mathrm{IND\text{-}CPA}}_{\mathcal{E},\mathcal{A}}(\lambda) = \Pr\left[\mathrm{IND\text{-}CPA}^{0}_{\mathcal{E},\mathcal{A}}(1^{\lambda}) = 1\right] - \Pr\left[\mathrm{IND\text{-}CPA}^{1}_{\mathcal{E},\mathcal{A}}(1^{\lambda}) = 1\right]$$

An encryption scheme $\mathcal{E}$ is $\mathrm{IND\text{-}CPA}$ secure if for all efficient adversaries $\mathcal{A}$ and the negligible function $\mathsf{negl}(\lambda)$, the adversary advantage is

$$\mathsf{Adv}^{\mathrm{IND\text{-}CPA}}_{\mathcal{E},\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda)$$

Definition 2.1.2 measures the adversaries's change of behavior from one experiment to the other.

2.1.2 Pseudorandom functions

A Pseudorandom Function (PRF) is a fundamental building block of cryptographic protocols. By itself a PRF does not ensure confidentiality but is used to analyse the security guarantees of more intricate protocols, such as key derivation, symmetric encryption schemes and message authentication. The prevailing approach in the provable-security paradigm is to analyse the security properties of a cryptographic system assuming the existence of a truly random function, i.e., a function chosen at random from a family of functions. Afterwards, the goal is to replace the random function with a function sampled from a family of functions that maintains similar security guarantees. The last step is possible using a PRF, a family of keyed function $F : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $\mathcal{K} = \{0,1\}^k$ is the set of all keys with size k, $\mathcal{D} = \{0,1\}^l$ is the set of all possible l-bit strings messages and $\mathcal{R} = \{0,1\}^L$ is the set of all possible output L-bit strings.

For a family of functions to be considered pseudorandom it has to satisfy two properties, efficient computation and security. A function can be efficiently computed if there is a polynomial algorithm that given as input a $\mathsf{sk} \in \mathcal{K}$ and a message $m \in \mathcal{D}$ it computes $F(\mathsf{sk}, m)$. A PRF is secure if it's indistinguishable from a random function. Intuitively, an adversary has access to a black-box that can not be inspected but can be queried. Given an input message the box outputs a random element, independent of any point in the function range. The only constraint is if the same input is provided multiple times than the same output must be returned. The adversary should not be able to discern if the output values are computed by a PRF or a random function. Formally, this definition is captured by a security game defined in Figure 2.2.

The security game considers an adversary that plays one of two possible experiments. Similar to the symmetric setting, the experiment is fixed by a random coin flip $b \leftarrow \{0,1\}$ not

Experiment $\mathrm{PRF}^0_{F,\mathcal{A}}(1^\lambda)$	Experiment $\mathrm{PRF}^1_{F,\mathcal{A}}(1^\lambda)$
$\mathsf{sk} \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$	$g \leftarrow\!\!\$\ \mathrm{FUNC}(\mathcal{D}, \mathcal{R})$
$b \leftarrow\!\!\$\ \mathcal{A}^{F_{\mathsf{sk}}}$	$b \leftarrow\!\!\$\ \mathcal{A}^{g}$
return b	**return** b

Figure 2.2: Game definition of pseudorandom function security.

disclosed to the adversary. Additionally, the adversary can query an oracle with a polynomial number of messages $m \in \mathcal{D}$ and receives an output $o \in \mathcal{R}$ for each query. However, in experiment $\mathrm{PRF}^0_{F,\mathcal{A}}$ the oracle computes the query result with a random instance of the F family by sampling a key sk from a key generation algorithm KGen. The key is never disclosed to the adversary and the oracle access is denoted as $A^{F_{\mathsf{sk}}}$. In experiment $\mathrm{PRF}^1_{F,\mathcal{A}}$ the oracle uses a random function g sampled from $\mathrm{FUNC}(\mathcal{D}, \mathcal{R})$, the family of all functions from $\mathcal{D}$ to $\mathcal{R}$. The oracle access is denoted as $\mathcal{A}^g$. Eventually, the adversary outputs a bit b that determines its guess on which experiment was executed.

Definition 2.1.3 (PRF Security)**.** Let F be a pseudorandom function, g a random function and $\mathcal{A}$ a PPT adversary. The PRF advantage of $\mathcal{A}$ is defined as follows:

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

A PRF is said to be secure if for all efficient adversaries $\mathcal{A}$ and the negligible function $\mathsf{negl}(\lambda)$, the adversary advantage is

$$\mathsf{Adv}^{\mathrm{PRF}}_{F,\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda)$$

2.2 Symmetric property-preserving encryption schemes

2.2.1 Deterministic encryption

A Deterministic Encryption Scheme (DET) is used to enable efficient, confidential searches in a two-party setting. In this setting, a storage party has an encrypted database of documents where each document is indexed with a unique tag. A client party wants to search for a exact match of a document indexed by a specific keyword from the storage party. However, the search has to be made in constant-time and the storage party cannot learn the contents of the keyword sent by the client party. Deterministic encryption schemes solve this problem by always outputting the same ciphertext for every encryption of a given plaintext message and secret key. Using this scheme, documents tag as well as keyword in client queries are encrypted under the same secret key. It is assumed the secret key is never disclosed to the storage party. If this assumption

Experiment $\text{IND-DCPA}^0_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Experiment $\text{IND-DCPA}^1_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Oracle $\mathsf{Encrypt_b}(msg_0, msg_1)$
$\mathsf{sk} \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$	$\mathsf{sk} \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$	**if** $\lvert msg_0\rvert \neq \lvert msg_1\rvert$ **do**
return $\mathcal{A}^{Encrypt_0}(1^\lambda)$	**return** $\mathcal{A}^{Encrypt_1}(1^\lambda)$	**return** $\perp$
		return $\mathsf{Enc}(\mathsf{sk}, msg_b)$

Figure 2.3: Game definition of deterministic encryption scheme security. This game is similar to the IND-CPA but in this experiment the encryption scheme $\mathcal{E}$ is deterministic. Furthermore, the adversary must send sequences of not repeatable messages.

holds, then the client sends a ciphertext to the storage party which only has to search for the document indexed by a tag equal to the client's ciphertext.

More precisely, a symmetric encryption scheme $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is a deterministic encryption scheme if the encryption algorithm $\mathsf{Enc}(\mathsf{sk}, \cdot)$ is a deterministic function for all keys sampled by the key generation algorithm $\mathsf{sk} \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$. A function $f : D \to R$ is a deterministic function if $\forall i, j, \in D, \quad f(i) == f(j) \leftrightarrow i = j$. A deterministic encryption scheme must also satisfy the correctness property of a symmetric encryption scheme.

Security is defined by the security game Indistinguishability under distinct chosen-plaintext attack (IND-DCPA) [BKN04] defined in Figure 2.3. Similar to the IND-CPA game, the adversary interacts with an oracle that hides the computation of the encryption scheme. But in IND-DCPA the oracle uses a deterministic encryption scheme to encrypt one of two input messages. Furthermore, the adversary has its powers limited and can only query the oracle with non repeatable messages. More precisely, a sequence of queries with size N sent to the oracle $(m_0^1, m_1^1), \ldots, (m_0^N, m_1^N)$ by the adversary $\mathcal{A}$ must have equal length $|m_0| = |m_1|$ messages and satisfy the following *message uniqueness* property:

$$\forall i, j, \quad 0 \leq i, j \leq N, \quad m_0^i \neq m_0^j \wedge m_1^i \neq m_1^j$$

Definition 2.2.1 (IND-DCPA Security)**.** Let $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a deterministic encryption scheme and $\mathcal{A}$ a PPT adversary with access to oracle $\mathsf{Encrypt_b}$. The IND-DCPA advantage of $\mathcal{A}$ is defined as follows:

$$\mathsf{Adv}^{\text{IND-DCPA}}_{\mathcal{E},\mathcal{A}}(\lambda) = \Pr\left[\text{IND-DCPA}^0_{\mathcal{E},\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\text{IND-DCPA}^1_{\mathcal{E},\mathcal{A}}(1^\lambda) = 1\right]$$

An encryption scheme $\mathcal{E}$ is $IND - DCPA$ secure if for all efficient adversaries $\mathcal{A}$ and the negligible function $\mathsf{negl}(\lambda)$, the adversary advantage is

$$\mathsf{Adv}^{\text{IND-DCPA}}_{\mathcal{E},\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda)$$

2.2.2 Order-preserving encryption

An Order Preserving Encryption (OPE) scheme considers a two-party setting similar to deterministic encryption schemes. However, the client party wants to search for a range of documents instead of a single exact document. Furthermore, the encryption scheme must also enable efficient searches in the storage party. Efficiency in this context is defined in logarithmic time, or sub-linear at least, in proportion to the number of stored documents. While neither deterministic or classic symmetric encryption schemes can satisfy these requirements, order-preserving encryption schemes address this issue by preserving the numerical order of the plaintext space in encrypted ciphertexts [Bol09].

The formal definition of an order-preserving scheme is related to the definition of an order-preserving function. For a numerical domain A and a numerical range B, such that $A, B \in \mathbb{N}$ and $|A| \leq |B|$ a function $f : A \to B$ is said to be *order-preserving* if $\forall i, j, \quad f(i) > f(j) \leftrightarrow i > j$. A symmetric encryption scheme $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ is an order-preserving encryption scheme if the encryption algorithm $\mathsf{Enc}(\mathsf{sk}, \cdot)$ is an *order-preserving* function for all secret keys sk sampled by the key generation algorithm $\mathsf{sk} \leftarrow\!\!\$\, \mathsf{KGen}$. Order-preserving schemes are also correct in the sense of symmetric encryption schemes.

An order-preserving encryption scheme can not be considered secure in standard security notions, such as IND-CPA or IND-DCPA, as the order between plaintexts is leaked. Instead, we present the security definition proposed by *Boldyreva et al.* [Bol09] for stateless order-preserving schemes. This security definition is more closely related to the security definitions of PRF, where an adversary has to distinguish between a real execution of an encryption scheme and an idealized order-preserving function. However, the security definition is slightly more powerful than the PRF definition, as the adversary has access to either the decryption algorithm or an inverse function of the order-preserving function. This definition is captured by the security game Pseudorandom order-preserving function against chosen-ciphertext attack (POPF-CCA) defined in Figure 2.4.

The security game of POPF-CCA consists on an adversary $\mathcal{A}$ that interacts with one of two experiments. The game starts by flipping a coin $b \leftarrow\!\!\$\, \{0, 1\}$ that fixes one of the experiments. In experiment $\mathrm{POPF\text{-}CCA}^{0}_{\mathcal{E},\mathcal{A}}$, the experiment starts by sampling a secret key sk using a key generation algorithm KGen from the order preserving encryption scheme $\mathcal{E}$. Afterwards, the adversary can send a polynomial number of queries to either an encryption oracle $\mathsf{Enc}_{\mathsf{sk}}$ or a decryption oracle $\mathsf{Dec}_{\mathsf{sk}}$. The encryption oracle encrypts plaintext messages with the secret key sk and outputs ciphertext messages; The decryption oracle does the reverse process. In experiment $\mathrm{POPF\text{-}CCA}^{1}_{\mathcal{E},\mathcal{A}}$, the experiment does not sample a secret key but instead samples a pair of ideal order-preserving functions g, g^{-1} from the set of all order-preserving functions $\mathrm{OPF}(\mathcal{D}, \mathcal{E})$ with domain $\mathcal{D}$ and range $\mathcal{E}$. The functions are exposed to the adversary as oracles $\mathcal{A}^{g}$ and $\mathcal{A}^{g^{-1}}$ that can be queried similarly to the previous experiment. At the end of the game the adversary outputs its guess on which experiment was fixed as a bit b'.

Experiment $\mathrm{POPF\text{-}CCA}^{0}_{\mathcal{E},\mathcal{A}}(1^{\lambda})$	Experiment $\mathrm{POPF\text{-}CCA}^{1}_{\mathcal{E},\mathcal{A}}(1^{\lambda})$
$\mathsf{sk} \leftarrow\!\!\$\ \mathsf{KGen}(1^{\lambda})$	$g, g^{-1} \leftarrow\!\!\$\ \mathrm{OPF}(\mathcal{A}, \mathcal{B})$
$b \leftarrow\!\!\$\ \mathcal{A}^{\mathsf{Enc}_{\mathsf{sk}},\mathsf{Dec}_{\mathsf{sk}}}$	$b \leftarrow\!\!\$\ \mathcal{A}^{g,g^{-1}}$
return b	**return** b

Figure 2.4: Game definition of order-preserving encryption scheme security.

Definition 2.2.2 ($\mathrm{POPF\text{-}CCA}$ Security)**.** Let $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric order-preserving encryption scheme and $\mathcal{A}$ a PPT adversary. The $\mathrm{POPF\text{-}CCA}$ advantage of $\mathcal{A}$ is defined as follows:

$$\mathsf{Adv}^{\mathrm{POPF\text{-}CCA}}_{\mathcal{E},\mathcal{A}}(\lambda) = \Pr\left[\mathrm{POPF\text{-}CCA}^{0}_{\mathcal{E},\mathcal{A}}(1^{\lambda}) = 1\right] - \Pr\left[\mathrm{POPF\text{-}CCA}^{1}_{\mathcal{E},\mathcal{A}}(1^{\lambda}) = 1\right]$$

An encryption scheme $\mathcal{E}$ is $\mathrm{POPF\text{-}CCA}$ secure if for all efficient adversaries $\mathcal{A}$ and the negligible function $\mathsf{negl}(\lambda)$, the adversary advantage is

$$\mathsf{Adv}^{\mathrm{POPF\text{-}CCA}}_{\mathcal{E},\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda)$$

2.2.3 Searchable encryption

Symmetric searchable encryption (SSE) schemes consider a trusted client that wants to outsource a private database of documents to an untrusted server without limiting its ability to query the data. Additionally, the untrusted server should not be able to infer the contents of the stored data or the queries issued. This problem closely resembles the issue addressed by DET and OPE schemes but SSE focus on defining encrypted indexes and search algorithms instead of proposing only algorithms for encryption and decryption. An SSE scheme creates specialized indexes that map keywords to documents. All of the contents of the index are encrypted and can be stored on the server side. The client can retrieve a subset of documents by sending a query to the server. However, the server can only correctly scan the index if the client generates trapdoors that disclose some information about the query and the documents.

There are two possible approaches to build the secure index: i.) a static secure index that cannot be updated once initialized. ii.) a dynamic secure index that supports inserts and deletions of documents and can be queried at any moment. An index may also support different types of queries, either single keyword search or multi-keyword boolean queries. We present a formal definition based on *Bost et al.* [BMO17] that captures the most general case of a dynamic searchable encryption scheme that supports multi-keyword boolean queries.

A plaintext searchable encryption database DB is defined as $\mathsf{DB} = \{(ind_i, W_i) : 1 \leq i \leq D\}$ where $D = |\mathsf{DB}|$ is the number of documents in the database. Each document is indexed by

a pair of document indices $ind_i \in \{0,1\}^l$, represented by l-bit strings, and a set of keywords W_i in the document ind_i. Each keyword is a binary string of variable length. Each document is identified by its indice and an actual document is abstracted as an arbitrary bitstring. A database query is a boolean predicate composed by multiple keywords denoted by τ. A database search $\mathsf{DB}(\tau)$ returns the set of documents satisfying the query $\mathsf{DB}(\tau) = \{ind_i : \tau(W_i) = 1\}$.

Definition 2.2.3 (Searchable Symmetric Encryption Scheme)**.** A searchable symmetric encryption scheme $\mathcal{SE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is defined by the following three PPT algorithms:

$\mathsf{Setup}(\mathsf{DB}) \rightarrow (\mathsf{EDB}, \mathsf{sk}, \alpha)$**:** Probabilistic algorithm that takes as input a plaintext database DB. It outputs an encrypted database EDB, a secret key sk and a client internal state α.

$\mathsf{Search}(\mathsf{sk}, \alpha, \tau, \mathsf{EDB}) \rightarrow (I_w, \mathsf{EDB}')$**:** Search protocol between the client and the untrusted server. The client takes as input a secret key sk, its internal state α and a database query τ. The untrusted server takes as input the encrypted database EDB. The protocol returns a list of document identifiers I_w and an updated encrypted database EDB'.

$\mathsf{Update}(\mathsf{sk}, \alpha, op, in, \mathsf{EDB}) \rightarrow \mathsf{EDB}'$**:** Update protocol between the client and the untrusted server. The client takes as input a secret key sk, its internal state α, an operation op and an input in. The operation is either the insertion of a new document or the removal of an existing one $op \in \{Add, Delete\}$. The input contains a document index ind and a set of indexed keywords W. The server takes as input an encrypted database EDB. The protocol output is an updated encrypted database EDB'.

The execution of a searchable encryption scheme starts with the client initializing its internal state and outsourcing the plaintext database to the untrusted server with the Setup algorithm. Afterwards, the client can either search for a subset of documents or update the encrypted database at will. A SSE is correct if for every client query τ and for every database DB the following condition holds true

$$\mathsf{Search}(\mathsf{sk}, \alpha, \tau, \mathsf{EDB}) = \mathsf{DB}(\tau)$$

and the secret key sk, encrypted database EDB and client internal state α are initialized by the Setup protocol.

A secure SSE scheme minimizes the information disclosed by client queries, but there is an explicit assumption that some information is leaked. A leakage free construction can be obtained albeit at a significant performance cost. The security of a SSE scheme is captured in real-world versus ideal-world game. The security game includes *leakage functions* that precisely define the information disclosed by to the server. Intuitively, an adversary has to distinguish between a real-world and the ideal-world. In both worlds, the adversary can activate the client functions at will and learn the information disclosed by each operation.

Experiment $\mathrm{REAL}^{\mathcal{E}}_{\mathcal{A}}(1^{\lambda}, q)$

$\mathsf{DB} \leftarrow \mathcal{A}()$
$(\mathsf{EDB}, \mathsf{sk}, \alpha) \leftarrow \mathsf{Setup}(\mathsf{DB})$
$Transcript \leftarrow (\mathsf{DB}, \mathsf{EDB})$
for $k = 1..q$ **do**
  $Q_k = (type_k, param_k) \leftarrow \mathcal{A}(Transcript)$
  if $type_k = Update$
    $R_k \leftarrow \mathsf{Update}(\mathsf{sk}, \alpha, param_k, \mathsf{EDB})$
  else
    $R_k \leftarrow \mathsf{Search}(\mathsf{sk}, \alpha, param_k, \mathsf{EDB})$
  Append (Q_k, R_k) to $Transcript$
$b \leftarrow \mathcal{A}(Transcript)$
return b

Experiment $\mathrm{IDEAL}^{\mathcal{E}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(1^{\lambda}, q)$

$\mathsf{DB} \leftarrow \mathcal{A}()$
$(\mathsf{EDB}, \mathsf{sk}, \alpha) \leftarrow S(\mathcal{L}^{Setup}(\mathsf{DB}))$
$Transcript \leftarrow (\mathsf{DB}, \mathsf{EDB})$
for $k = 1..q$ **do**
  $Q_k = (type_k, param_k) \leftarrow \mathcal{A}(Transcript)$
  if $type_k = Update$
    $R_k \leftarrow S(\mathcal{L}^{Update}(param_k))$
  else
    $R_k \leftarrow S(\mathcal{L}^{Search}(param_k))$
  Append (Q_k, R_k) to $Transcript$
$b \leftarrow \mathcal{A}(Transcript)$
return b

Figure 2.5: Game definition of SSE security.

Formally, the security game is defined by the real world experiment $\mathrm{REAL}^{\mathcal{E}}_{\mathcal{A}}$ and the ideal world experiment $\mathrm{IDEAL}^{\mathcal{E}}_{\mathcal{A},\mathcal{S},\mathcal{L}}$ where the adversary is denoted by $\mathcal{A}$ and there is a simulator denoted by $\mathcal{S}$. The adversary has complete control of the query inputs and can observe the entire history of requests between the client and the server. Additionally, the adversary can also observe the server's internal state such as memory contents, access as well as the storage I/O. In the real world, the adversary observes an honest execution of the SSE scheme. In the ideal world, the experiment is parameterized by leakage functions $\mathcal{L}$ = ($\mathcal{L}^{Setup}$, $\mathcal{L}^{Search}$, $\mathcal{L}^{Update}$) which correspond to the real world protocols, Setup, Search and Update. In this world, the transcript of message is generated by the simulator S which is a PPT algorithm. The inputs to the simulator are the result of the respective leakage function when provided the client input, e.g.: $\mathcal{S}(\mathcal{L}^{Setup}(\mathsf{DB}))$. In both worlds, the adversary can adapt its attack strategy according to the results of previous queries. After a polynomial number of queries the adversary outputs a guessing bit b. If the adversary cannot distinguish between both worlds, then the SSE scheme is secure. The formal definitions of the experiments are defined in Figure 2.6.

Definition 2.2.4 (Adaptive security of SSE schemes)**.** A symmetric searchable encryption scheme $\mathcal{SE} = (\mathsf{Setup}, \mathsf{Search}, \mathsf{Update})$ is $\mathcal{L}$-adaptive-secure if for every PPT adversary $\mathcal{A}$ that issues a polynomial number of queries $q(\lambda)$ there exists a PPT simulator $\mathcal{S}$ such that

$$\left| \Pr\left[\mathrm{REAL}^{\mathcal{E}}_{\mathcal{A}}(\lambda, q) = 1\right] - \Pr\left[\mathrm{IDEAL}^{\mathcal{E}}_{\mathcal{A},\mathcal{S},\mathcal{L}}(\lambda, q) = 1\right] \right| = \mathsf{negl}(\lambda)$$

2.3 Secure function evaluation

2.3.1 Homomorphic encryption

Homomorphic encryption schemes enable the computation of functions on encrypted data without access to the secret key. We consider a setting where a trusted party outsources some computation to an untrusted party. The trusted party can send inputs to the untrusted party, which evaluates the outsourced computation. However, the trusted party wants to ensure that no information regarding the input values, the computation result or any intermediate result is disclosed to the untrusted party. There are two solutions that fall within the class of homomorphic encryption, partial homomorphic encryption schemes and fully homomorphic schemes. The first class of encryption schemes only supports the computation of a subset of functions [Pai99] while the latter can evaluate any function. We use the standard definition of fully homomorphic computation proposed by *Gentry* [Gen09] where functions are defined as arithmetic circuits. This definition considers a public-key setting with a trusted party that generates a pair of keys, a secret key and a public key. The trusted party shares with the untrusted party the public key, a circuit and inputs encrypted with the private key. The untrusted party can evaluate the circuit with the encrypted circuit using the public key and obtains an encrypted result. The scheme is secure as long as the secret key is never disclosed to the untrusted party.

Definition 2.3.1 (Homomorphic Encryption Scheme)**.** An asymmetric fully homomorphic encryption scheme $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Evaluate})$ is defined by the following four PPT algorithms:

$\mathsf{KGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$**:** Probabilistic key generation algorithm that given as input the security parameter 1^λ outputs a new secret key sk and a public key pk. The public key defines the plaintext space $\mathcal{P}$ and ciphertext space $\mathcal{C}$.

$\mathsf{Enc}(\mathsf{sk}, m) \to c$**:** Probabilistic encryption algorithm that takes as input a secret key sk and a plaintext message $m \in \mathcal{P}$. It outputs a ciphertext $c \in \mathcal{C}$.

$\mathsf{Dec}(\mathsf{sk}, c) \to m$**:** Deterministic algorithm that takes as input a secret key sk and a ciphertext $c \in \mathcal{C}$. The result of this algorithm is a decrypted plaintext message $m \in \mathcal{P}$.

$\mathsf{Evaluate}(\mathsf{pk}, C, \gamma) \to c$**:** Probabilistic algorithm that takes as input a public key pk, a circuit $C \in \mathcal{C}_\mathcal{E}$ and a tuple of L ciphertext $\gamma = \langle c_1, \ldots, c_L \rangle$. The evaluation of the circuit results in a ciphertext $c \in \mathcal{C}$.

For an encryption scheme to be fully homomorphic it must ensure three properties: correctness, compact evaluation and security. Informally, a correct $c \leftarrow \mathsf{Evaluate}(\mathsf{pk}, C, \gamma)$ function returns the encryption of evaluating a circuit C with some plaintext inputs, i.e., $c \leftarrow \mathsf{Enc}(\mathsf{sk}, C(\langle (m_1, \ldots, m_L) \rangle)$. However, this property alone is not sufficient to exclude trivial schemes that may simply compute the circuit in the Dec algorithm instead of the Evaluate algorithm. As such,

Experiment $\text{IND-CPA}^0_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Experiment $\text{IND-CPA}^1_{\mathcal{E},\mathcal{A}}(1^\lambda)$	Oracle $\mathsf{Encrypt_b}(msg_0, msg_1)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$	$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$	**if** $\lvert msg_0\rvert \neq \lvert msg_1\rvert$ **do**
return $\mathcal{A}^{Encrypt_0}(1^\lambda, \mathsf{pk})$	**return** $\mathcal{A}^{Encrypt_1}(1^\lambda, \mathsf{pk})$	**return** $\perp$
		return $\mathsf{Enc}(\mathsf{sk}, msg_b)$

Figure 2.6: Game definition of public key homomorphic encryption security.

a fully homomorphic scheme must have a compact evaluation, meaning that the Dec algorithm is upper-bounded to a polynomial size that depends on the security parameter. We formally define these properties as follows:

Definition 2.3.2 (Correctness)**.** A homomorphic encryption scheme $\mathcal{E}$ is correct for any circuit $c \in \mathcal{C}_\mathcal{E}$, for any pair of keys generated by the key generation algorithm $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathsf{KGen}(1^\lambda)$, for any $L-$tuple of plaintexts $\langle m_1, \ldots, m_L\rangle$ where $m_i \in \mathcal{P}$ and any ciphertexts $\langle c_1, \ldots, c_L\rangle$ such that $c_i \leftarrow \mathsf{Enc}(\mathsf{sk}, m_i)$,

$$c \leftarrow \mathsf{Eval}(\mathsf{pk}, C, \langle c_1, \ldots, c_L\rangle) \implies \mathsf{Dec}(\mathsf{sk}, c) = C(\langle m_1, \ldots, m_L\rangle)$$

Definition 2.3.3 (Compact Homomorphic Encryption)**.** A homomorphic encryption scheme $\mathcal{E}$ is compact if for every security parameter 1^λ, the decryption algorithm Dec can be expressed as a circuit $D_\mathcal{E}$ upper-bounded by the polynomial function $f(1^\lambda)$.

Definition 2.3.4 (Fully homomorphic encryption)**.** A homomorphic encryption scheme is fully homomorphic if its correct and compact for every circuit $c \in C_\mathcal{E}$.

The security of homomorphic encryption schemes is defined by a Indistinguishability under chosen-plaintext attack (IND-CPA) in a public-key setting, defined in Figure 2.6. This game is mostly similar to the symmetric indistinguishability game but has two main differences. First, the experiments samples a pair of private and public keys, and shares the public key with the adversary. Secondly, with access to the public key the adversary can run the Evaluate algorithm at will, for any circuit and sequence of ciphertexts. Besides these two changes the adversary still has access to an oracle that hides the encryption of plaintexts messages and it has to distinguish between the execution of two experiments.

Definition 2.3.5 (Homomorphic Encryption Security)**.** A public-key homomorphic encryption scheme $\mathcal{E} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Evaluate})$ is IND-CPA secure if for all efficient adversaries $\mathcal{A}$ and the negligible function $\mathsf{negl}(\lambda)$, the adversary advantage is

$$\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{E},\mathcal{A}}(\lambda) < \mathsf{negl}(\lambda)$$

2.3.2 Secure multiparty protocols

Secure multiparty computation (SMPC) considers a setting of multiple independent parties that want to compute an agreed function. Each party is willing to provide a private input to the computation, but they do not want to disclose their input to other parties. This privacy condition must hold even in the presence of malicious parties that may try to learn the private inputs. Furthermore, honest parties also want the computation output to be correct. A trivial solution to this problem is rely on a trusted third-party and use secure channels to send the inputs. The trusted third-party computes the agreed function and broadcasts the correct result. However, assuming the existence of a trusted third-party is not always feasible. As such, SMPC protocols address this problem without using a trusted third-party and remains secure even if there is a subset of corrupted parties.

SMPC is defined by a set of N parties $P = \{P_1, P_2, \ldots, P_N\}$, a vector of private inputs $\{x_1, \ldots, x_N\}$, a function f and a distributed protocol π. Each party P_i holds a single private input x_i and wants to compute the function $f(x_1, \ldots, x_N) = (y_1, \ldots, y_N)$ such that each party P_i learns y_i, but nothing else. The protocol π is a distributed computation that describes the steps each party takes to globally evaluate f and securely obtain its results. Protocols with more than two parties are usually generated by compiling a function f into either a boolean circuit or an arithmetic circuit which is evaluated at run-time by the parties. A circuit consists of multiple gates that are evaluated either with local steps or distributed computations. Regardless of the gate, the information processed by the circuit is always encrypted and only the function result is disclosed to the parties.

Security. The security of SMPC is analysed in a *real world* versus *ideal world*. Intuitively, the ideal world that captures the highest level of security and functionality by executing the computation in an trusted third-party. In the real world the trusted party does not exist and the desired functionality is computed by a distributed protocol π. A protocol π is said to be secure if an adversary cannot distinguish between the execution of both worlds. In more detail, we present a security definition of multi-party protocols in a semi-honest model [CD05, EKR18, Gol04]. The number of parties in the model is fixed and does not change between its execution. An adversary can corrupt a subset $I \subset P$ of parties and learn everything each party views during its execution. The set of corrupted parties is defined before the protocol starts and does not change during its execution. In this model we consider a passive adversary that only observes the protocols execution and does not (cannot) force a party to deviate from its execution.

In the *ideal world*, every party sends its own input x_i to a trusted third-party, which evaluates the desired function $f(x_1, \ldots, x_N) = (y_1, \ldots, y_N)$. The output result y_i is sent securely from the trusted third-party to each party P_i. In this world there is a simulator S that is given the inputs and outputs of the corrupted parties. This simulator captures the information that an adversary

learns during a protocol execution. The *ideal-world* is defined by the following distribution of random variables:

$$\text{IDEAL}_{I,\pi,S}(x_1,\ldots,x_N) = (S(\{(i,x_i,y_i)\}_{i\in I}), f(x_1,\ldots,x_N))$$

In the *real world* the parties evaluate a protocol π and each party has a view that is defined by its private input x_i, its random tape r_i and the sequence of all the messages exchanged m_i. The adversary also has its own view which is defined by the union of the views of all of the corrupted parties. Everything the adversary learns in the *real-world* must be efficiently computed from its view. Formally, we define the *real-world* by the following distribution of random variables:

$$\text{REAL}_{I,f}(x_1,\ldots,x_N) = (\{(i,x_i,y_i,msg_i;r_i)\}_{i\in I}, f(x_1,\ldots,x_N))$$

Definition 2.3.6 (Semi-honest security)**.** We say that a protocol π securely computes f if for all inputs $x_i,\ldots,x_n$, for every PPT adversary $\mathcal{A}$, for all $I \subset P$ there is a PPT simulator S such that:

$$\text{IDEAL}_{I,\pi,S}(x_1,\ldots,x_N) \approx \text{REAL}_{I,f}(x_1,\ldots,x_N)$$

where $\approx$ denotes that the two distributions can either be computationally indistinguishable, statistically indistinguishable or identical, i.e., the protocol is perfectly secure. If the *real-world* execution can be simulated from only the inputs and outputs of the corrupted parties, then the adversary does not learn any more information from the real-world than from an ideal execution. This security definition also implicitly captures the correctness property of SMPC protocols as it requires that the output of the real world to be equal to the ideal world. Otherwise, the adversary can trivially distinguish between both worlds.

Sharemind protocols for addition and multiplication

We now provide a better intuition on how complex Secure multiparty computation (SMPC) can be constructed by presenting two atomic components of a concrete Secure multiparty computation (SMPC) framework. More concretely, we describe the addition and multiplication gates of the Sharemind framework [Bog13]. These atomic operators are extensively used to build protocols that compare the equality and order of private information. These protocols are limited to three parties and consider a semi-honest adversary that can corrupt at most a single party.

The Sharemind framework, similar to existing work in the state-of-the-art, uses a secret sharing scheme as the main tool to build the protocols. A secret sharing scheme is a cryptographic protocol that splits a private input in N shares, such that each share does not disclose any information on the private input. The original private input can only be reconstructed if more than t shares are brought together in a single party. Formally, a secret sharing scheme

$\mathcal{S} = (\mathsf{Encode}, \mathsf{Decode})$ is defined by two PPT algorithms. The $\mathsf{Encode}(m) \rightarrow (s_1, \ldots, s_N)$ algorithm takes as input a plaintext message m and outputs a $n-$tuple of shares. The $\mathsf{Decode}(s_1, \ldots, s_N) \rightarrow m$ is the inverse function, it takes as input an $n-$tuple of shares and outputs a plaintext message m. A secret sharing scheme $\mathcal{S}$ is correct if the following condition holds $m = \mathsf{Decode}(\mathsf{Encode}(m))$.

Sharemind uses an additive secret sharing scheme over the finite ring $\mathbb{Z}_{2^{32}}$. We denote by $u \in \mathbb{Z}_{2^{32}}$ a private input message and by $\bar{u} = (u_1, u_2, u_3)$ a tuple with the shares of u, such that the share s_i is held by the party P_i.

One of the advantages of using additive secret sharing is the computation of the addition gates, which does not require any communication between the parties. Given two private inputs u and v, and its shares $\bar{u}$ and $\bar{v}$, each party P_i has access to the share u_i and v_i. To evaluate the addition gate, each party only needs to do a local computation and sum its shares. As such, the addition gate can be defined as follows:

$$\bar{u} + \bar{v} = \sum_i^3 u_i + v_i \mod 2^{23}$$

The evaluation of the multiplication gate requires a few extra steps and parties have to exchange messages. Intuitively, the multiplication of two secret shared u and v is the dot product of its shares such that $w = u \cdot v = (u_1 + u_2 + u_3) \cdot (v_1 + v_2 + v_3)$. By expanding the dot product, it's clear that each share u_i multiplies with every share v_i. However, for the protocol to be secure the parties can't simply send all their shares to each other as it would allow the corrupted party to disclose the input values. Instead, the Sharemind protocols forwards a single secret u_i and v_i to the next party $P_{(i+1 \mod 3)}$ and locally computes w_i as follows:

$$w_i = u_i \cdot v_i + u_i \cdot v_{(i+2 \mod 3)} + u_{(i+2 \mod 3)} \cdot v_i$$

2.3.3 Isolated execution environments

An Isolated Execution Environment (IEE) is an abstraction that captures the security properties of trusted hardware solutions such as Intel SGX and Trust Zone [MAB+13]. These systems address a problem similar to homomorphic encryption, where a trusted party wants to outsource the computation of some function to an untrusted party without compromising its confidentiality. However, in this setting the function is a stateful program and the untrusted party provides access to an IEE. The IEE can execute any program within a protected environment and ensure the integrity and confidentiality of the program's internal state. The following definitions are based on the formal treatment of secure outsourced computation proposed by Barbosa et al. [BPSW16].

The problem of outsourcing computation to a trusted hardware is defined by a program P executed in a machine $\mathcal{M}$. The program is a transition function that is activated with a state

st and an input i and returns an output o. Furthermore, the program can also take as input a source of randomness r. A sequence of program activations is denoted as $(o_1, \ldots, o_n) \leftarrow P[st; r](i_1, \ldots, i_n)$ and $Trace_{P[st;r]}(i_1, \ldots, i_n)$ denotes the I/O trace of a program execution $(i_1, o_1, \ldots, i_n, o_n)$. The machine $\mathcal{M}$ is a Random Access Machine (RAM) with its behavior defined by the program execution and its inputs. The internal state of the machine is inaccessible to any external programs and the only information disclosed are its inputs and outputs.

Definition 2.3.7 (IEE Machine)**.** An IEE Machine $\mathcal{M}$ is defined by the following interface:

$\mathsf{Init}(1^\lambda) \rightarrow prms$**:** Procedure that initializes the machine security hardware. It takes as input the security parameter and outputs some global public parameters $prms$.

$\mathsf{Load}(P) \rightarrow hdl$**:** Procedure that initializes a single IEE instances with a program P. It outputs an IEE handle that is used to send inputs to a specific running instance.

$\mathsf{Run}(hdl, i) \rightarrow o$**:** Procedure that activates a program being executed in an IEE with the handle hdl. The input i is passed on the program within the IEE which returns an output o.

By itself, an IEE is not sufficient to protect the confidentiality of a program's remote execution. It only provides a strict isolation between functions running in different IEEs as well from any other external context in a machine. However, it does not provide any confidentiality or integrity of the inputs and outputs of the execution.

Secure Outsourced Computation

For a program P to be securely evaluated in a remote machine $\mathcal{M}$ it needs to satisfy the security definition of Secure Outsourced Computation (SOC) protocols. Intuitively, a program is securely evaluated on a remote machine if the remote view of the IEE is identical to a local execution and the messages exchanged between the client and the remote machine are confidential. In fact, a SOC protocol is not limited to IEEs and instead builds on the general notion of attested computation where a trusted party has to *attest* that a program P is running on a trusted environment and that the remote I/O trace of P execution matches the I/O trace of an honest execution. If a remote machine provides attestation than a key exchange protocol can be used to establish a secure communication channel with the trusted environment (e.g.: an IEE) that protects the confidentiality of inputs and outputs.

Definition 2.3.8 (Secure Outsourced Computation)**.** A secure outsourced computation scheme $\mathcal{E} = (\mathsf{Compile}, \mathsf{BootStrap}, \mathsf{Verify}, \mathsf{Encode}, \mathsf{Attest})$ for a remote machine $\mathcal{M}$ is defined by the following PPT algorithms:

$\mathsf{Compile}(prms, P, id) \rightarrow (P^*, st)$**:** Compilation algorithm that takes as input the public parameters $prms$, the program P and party identified id. It returns a client side state st and a

Experiment $\mathrm{PRIV}_{\mathcal{E},\mathcal{A}}(1^\lambda)$

$id \leftarrow \{0,1\}^*$
$prms \leftarrow\!\!\$\ \mathcal{M}.Init(1^\lambda)$
$P \leftarrow\!\!\$\ \mathcal{A}(prms)$
$(P^*, st) \leftarrow\!\!\$\ \mathsf{Compile}(prms, P, id)$
return $\mathcal{A}^{Send_b, Bootstrap}(P^*)$

Oracle $Send_b(o^*, i_1, i_2)$

$(o, st) \leftarrow \mathsf{Verify}(prms, o^*, st)$
if $|i_1| \neq |i_2|$ **do**
$(i^*, st) \leftarrow \mathsf{Encode}(prms, i_b, st)$
return i^*

Oracle $BootStrap(o)$

if $st.accept = T$ **do**
  return $\perp$
$(i, st) \leftarrow \mathsf{BootStrap}(prms, o, st)$
return i

Figure 2.7: Input privacy of SOC scheme.

compiled program P^* instrumented with a remote key attestation algorithm. The trusted client sets an initial flag $st.accept = \perp$.

$\mathsf{BootStrap}(prms, o, st) \rightarrow i$**:** Trusted party initialization algorithm that takes as input the public parameters $prms$, output o and an local state st. It runs a local key exchange algorithm that returns the next input of the bootstrap phase i to be delivered to the IEE. Bootstrap is complete when the $st.accept = true$.

$\mathsf{Encode}(prms, i, st) \rightarrow (i^*, st')$**:** Trusted party encoding algorithm that takes as input the public parameters $prms$, the local state st and an input i for a program P. It outputs an encrypted input i^* for program P^* as well as an updated state st'.

$\mathsf{Attest}(prms, hdl, i^*) \rightarrow o^*$**:** Untrusted party attestation algorithm that takes as input the public parameters $prms$, the program handle hdl and a program input i^*. It invokes the machine $\mathcal{M}.Run(hdl, i)$ and outputs an attested output o^*.

$\mathsf{Verify}(prms, o^*, st) \rightarrow b$**:** Trusted party verification algorithm that is given as input the public parameters $prms$, the local state st and an attested output o^*. It validates the authenticity of the output and returns a boolean result b.

To outsource a program P, the trusted party starts by compiling the program and obtaining a program P^* to be loaded on to a machine $\mathcal{M}$. After loading the program, the trusted party cannot immediately activate it with new inputs. Instead, there is a bootstrap phase between both parties which consists of an attested key exchange protocol. If the bootstrap phase is successful, then the trusted client shares a secret state with the program P^* loaded on a trusted IEE. The trusted client can then activate P by encoding the messages and sending them using the Attest protocol. The authenticity of the activation output is verified with Verify protocol.

An SOC scheme is secure if it ensures *input integrity* and *input privacy*. Intuitively, the *input integrity* property states that the local view of a trusted party and the remote view of a program P^* must coincide. The views can only differ in the last message which might not have been delivered.

Definition 2.3.9 (Input Integrity)**.** A SOC scheme $\mathcal{E}$ satisfies *input integrity* if for every program P and input i the remote I/O trace T is equal to the local I/O trace T',

$$\psi(T, T') := T = T' \vee \exists o.(T = o :: T')\exists i.(T' = i :: T)$$

The *input privacy* property defines that an untrusted party can not disclose the contents of the trusted party inputs. Similar to symmetric encryption schemes, *input privacy* is defined with an indistinguishability game PRIV where an adversary is given access to an oracle that returns the encryption of one message from a pair of inputs. An SOC scheme is considered secure if an adversary cannot distinguish which message was encrypted. The indistinguishability game is formally defined in Figure 2.7.

Definition 2.3.10 (Input Privacy)**.** Let, $\mathcal{E}$ be secure outsourced computation scheme, $\mathcal{M}$ and $\mathcal{A}$ a PPT adversary. *Input privacy* is defined as follows:

$$\left| \Pr\left[\mathrm{PRIV}^0_{\mathcal{E},\mathcal{A}}(\lambda) = 1\right] - \Pr\left[\mathrm{PRIV}^1_{\mathcal{E},\mathcal{A}}(\lambda) = 1\right] \right| = \mathsf{negl}(\lambda)$$

Definition 2.3.11 (SOC Security)**.** A secure outsourced computation scheme is secure if it satisfies *input privacy* and *input integrity*.

2.4 Summary

The problem of outsourcing private data and computation requires a thorough analysis of the existing cryptographic schemes and their security properties. In this section we reviewed a few cryptographic building blocks that ensure data confidentiality and secure function evaluation. Conceptually, these schemes can be combined to provide a general solution to the problem of CPD. In a two-party setting it is viable to store sensitive data on an untrusted by encrypting data with a symmetric encryption scheme and use one of the secure function evaluation techniques to extract some information. In fact, existing similar state-of-the-art solution follow a similar approach albeit at significant performance cost that requires either compromising some partial information and/or limit the computation that can be evaluated on the untrusted side. These trade-offs are explored in the following chapters.

Chapter 3

Related work

The existing work towards cryptographic protected databases has been a gradual process with roots on system research and cryptography. In this chapter we overview the main contributions that have merged both research fields, starting on secure data storage solutions and moving to general-purpose CPD. For each class of system presented we also outline the functionality supported and the expected performance (asymptotic or practical). However, before we present the state-of-the-art, we detail different types of threat models which are used by the systems to capture the powers of an adversary and the information the system aims to protect.

3.1 Adversary model

CPDs are generally modeled in a two-party setting where one party is corrupted by an adversary. In this setting, one party is a client residing on a trusted site and the other is a backend service hosted on an untrusted site. The client captures a party that either owns a private dataset or is responsible for assuring its confidentiality. The client leverages the storage and computational resources provided by the backend service. Conceptually, both parties can be as powerful or as thin as necessary. Furthermore, there is no restriction on the number of nodes that constitute a party as they can scale horizontally as necessary. For instance, one possible system has a thin client where all of the data storage and processing is offloaded to a third-party service such as a cloud provider. Conversely, it is also feasible to have a system where the client is a private cluster with sufficient resources to handle high-throughput workloads and the backend is a public-cloud service used exclusively for replicated or archival storage.

The adversary corrupts the untrusted party to learn the data stored as well as the computation evaluated by the backend service. Systems can tolerate some adversaries and remain secure even after the untrusted party become corrupted. The type of attacker supported by a CPD is defined by the *adversary model* that defines the methods used to corrupt the untrusted site, the

power of the attacker and its actions after a successful corruption. The method used to corrupt the backend defines the adversary as either an *internal attacker* or an *external attacker*. An *internal attacker* has explicit access and control over the backend service, similar to a cloud or database administrators. The *external attacker* does not have direct access to the backend. Instead, the adversary attempts to either compromise the communication between the client and the backend or gain access to the backend service by exploring any existing vulnerabilities.

A system with a corrupted backend can still be secure depending on the assumed *adversary power*, i.e., the actions taken by the adversary to extract relevant information from a cryptographic protocol. Some systems only consider adversaries that passively observe messages exchange and data stored while other systems are secure against active attempts to modify the protocol execution. Depending on its power, an adversary can be classified as follows:

Semi-honest adversary: Attempts to compromise the protocol's privacy by only gathering information during the execution of the protocol. The adversary does not attempt to modify or hide messages exchanged between the parties.

Active adversary: Takes active actions such as sending messages that differ from an honest protocol execution or corrupt the messages received by a party in an attempt to learn some additional information.

The backend service can have a single or multiple nodes, each storing and processing only a subset of the total data. Even if the adversary gains control over subset of nodes it does not necessary imply that the system's confidentiality is compromised. For instance, SMPC protocols are designed to remain secure until the number of corrupted nodes reaches an established threshold, for instance the majority of the nodes. After the threshold of corrupted parties, a protocol no longer guarantees privacy. The adversary model also defines when parties become corrupted and how the set of corrupted parties changes during a protocol execution [Gol04]:

Static: The adversary corrupted a fixed (arbitrary) set of nodes before the protocol execution start. No additional nodes can be corrupted or become honest after the initialization.

Adaptive: The adversary has the power to corrupt any node during the execution of the protocol using the information it gathers. However, the number of corrupted nodes has to be within the upper bound and a corrupted party cannot become honest.

Mobile: Similar to an active adversary however, the set of nodes corrupted changes dynamically during the execution, i.e., a corrupted node can become honest.

3.2 Remote encrypted file systems

A straightforward solution to safely store sensitive data on an untrusted site is to use remote encrypted file systems. These systems abstract a remote storage service by exposing a local file system interface identical to a native file system. On the trusted site, client applications can relay requests to the remote storage service on an untrusted site by interacting with the abstract local interface. However, before a request is sent from the trusted site to the untrusted site they are protected. More concretely, every write request is intercepted and the plaintext data in the request is encrypted with a symmetric encryption scheme. Read requests are also intercepted and the data contents decrypted before being displayed to the client. Database systems can leverage remote encrypted file systems to outsource data and fetch only the data necessary to process queries without having to manage remote connections, file system logic or explicitly encrypting data. Most remote encrypted file systems in the related-work do not explicitly define the *adversary model* but the untrusted site is assumed to be corrupted by a semi-honest, static adversary. While there a few systems that support active adversaries by either using integrity checks (checksum) such as SUNDR or authenticated encryption such as SafeSky [KM17], the main goal of remote encrypted file systems is to ensure data confidentiality.

One of the main challenges of encrypted file systems is finding the appropriate abstraction level to intercept the client I/O requests on the trusted site. Existing systems assume the trusted site has a modern operating system architecture divided between the *kernel space* and the *user space* [SGG12]. The access to a storage device is abstracted by a virtual file system in the *kernel space*. Client applications reside on the *user space* and cannot send requests directly to the virtual file system. Instead, applications in the *user space* access the underlying storage by issuing system calls that forward requests to the *kernel space*. With this division, there are two approaches in the related work. One approach is to push down and delegate the encryption process to the operating system. Client applications can use the underlying file system without any modification besides some initial configuration. However, this approach requires client applications to relinquish some flexibility and control over how data is protected. The other option, in the opposite direction, is to move the encryption process as close as possible to the client application. Applications are no longer bound by the security guarantees provided by the operating system and can use a cryptographic system suitable to their use-case. However, custom solutions are prone to security vulnerabilities if not implemented correctly.

Kernel space solutions. Encrypted file systems embedded at the operating system level rely on a *vnode* interface to intercept I/O requests to physical files. This interface is a Unix kernel space abstraction that is placed between the virtual file system and a physical file system, such as ext4. A single *vnode* is a logical representation of an active open file, directory or socket that encapsulates the low-level file system details from higher-level operating system components. During a system execution, system calls are translated into virtual file system calls

which forwards them to a specific vnode invocation. Kernel space solutions add an additional layer between the virtual file system and the physical file system vnode.

Kernel space solutions have several advantages in terms of security, performance and interoperability. One immediate advantage is the transparent integration with the operation system from the client application perspective as system calls and the virtual file system interface is not modified. Furthermore, these systems are also portable as they can leverage the vnodes interface of different physical file systems. Regarding performance, one clear advantage is the file-level granularity which can be used to encrypt only a subset of sensitive data instead of the entire file system. There is also is no overhead on context switch between user space and kernel space. Finally, the kernel-space is protected from user space applications which cannot gain direct access to its internal memory.

CryptFS [ZBS98] was the first system to propose a kernel space solution. This system can be mounted as directory of the file system such that any file stored in the mount point is transparently encrypted before being stored. The system only supports a single encryption scheme, Blowfish [Sch93], and has a limited key management. Both eCryptFS [Hal10] and NCryptFS [WMZ03a] improve upon these limitations and extend CryptFS functionality. In particular, eCryptFS extends CryptFS with encryption polices and a key management that associate cryptographic meta-data (encryption keys) to each file instead of an entire mount point. Additionally, it leverages an IEE to bind a set of files to a specific machine and securely manage secret keys. NCryptFS provides a more general solution than CryptFS by supporting multiple users, multiple encryption schemes and authentication methods.

User space solutions. User space file systems can be implemented at two different levels of the I/O stack. The first option is to intercept an application request before leaving the application logic by swapping the standard shared I/O library with a secure library that has the same interface. In this approach, data is encrypted before leaving the application and a system call is made. However, files are encrypted on a per application basis and not every application interacts with the file system through a shared library. The second option is to intercept requests in the kernel space and redirect them back to a user space file system framework. Generally, the framework eases the development and prototyping of novel file systems. This class of frameworks have a kernel space component that redirect virtual file system requests to a user space daemon. The user space daemon implements the logic of the encrypted file system. This division between components isolates the user space daemon from the low-level details of the physical file systems but require multiple context switches between the user space and kernel space. Currently there are two main frameworks that follow this approach, the Network File System (NFS) [SGK+85] and Filesystems in Userpace (Fuse) [Mik05].

User space file systems have one clear advantage over kernel space file systems, a faster development cycle and a wider range of standard libraries. Kernel space encrypted file systems

are limited to encryption services provided by the kernel. Even though it is conceptually possible to migrate a user space library to the kernel it is in practice a complex task, often prohibitive due to security concerns. A single bug is sufficient to freeze the entire execution of the system. User space file systems have none of these limitations, a software error only stops the execution of single process, and there is a wide range of standard cryptographic libraries (e.g.: OpenSSL, PGP), debug and profiling tools available. Cloud services further enhance the demand of user space encrypted file systems as the only way for applications to interact with cloud environments is through user space middleware libraries.

Encrypted file systems first started with a user space approach, CFS [Bla93]. This system was limited to single host, a local encrypted file system, and used a redirection approach based on NFS. Client applications used a regular NFS client to interact with the file system but when a requested reached the kernel space they were redirected through the localhost network interface back to a user space NFS server daemon. This NFS server was extended to encrypt all of the files under a single directory with the same cryptographic key. TCFS follows a similar approach to CFS but supports remote encrypted file systems and has an extended key management [CCSP01]. One of the main drawbacks of NFS based user space file systems is the overhead of unnecessary network requests on a single host. The Fuse framework provides an optimized solution for user space file system by providing an in-kernel driver that is registered with the virtual file system as a regular physical file system. However, I/O requests to the Fuse driver are forwarded back to a user space daemon through system calls. This framework has been widely used to develop over 100 systems [TGS+15], prototypes and complete remote encrypted file systems such as EncFS [enc], CryFS [cry] and LessFS [les]. Both NFS and Fuse as well as shared libraries have been used to outsource storage to cloud environments with systems such as BlueSky [VSV12] and SCFS [BMO+14] that abstract the underlying cloud, ensuring strong consistency and high-availability. These guarantees are provided even in multi-client environments that share the same cloud service. Depsky [BCQ+13] goes a step further with dependable storage in a cloud-of-clouds setting which leverages multiple clouds to prevent loss and data corruption, system downtime and avoid vendor lock-in.

Data encryption at rest. Clearly, databases deployed on a private client infrastructure can use encrypted file system to transparently outsource private data to a remote untrusted server. However, it is important to mention that enterprise-grade database systems provide an alternative solution, encryption at rest. MySQL [mys] and Oracle [ora] are two of the leading commercial databases that provide this feature. With this solution, client applications transmit their data as plaintext to the database. Usually, data is only encrypted when persisted on the file system and is decrypted back to the main memory to process queries. A semi-honest adversary with access to the database engine can intercept client requests and learn sensitive information. As such, this solution only provides security in case the storage device is stolen or its contents leaked.

3.3 Encrypted indexes with controlled leakage

Encrypted databases can offload both data storage and query computation to an untrusted backend service by using searchable encryption schemes. The state-of-the-art is brimming with novel constructions that cover divergent security requirements. This section focus on the single writer, single reader setting as it is the best fit for encrypted databases and has seen the highest number of contributions [KS14b]. In this setting, the client is the sole owner of the database and it is the only party that can send queries. As defined in Section 2.2.3, SSE schemes encrypt the data on the client side before storing it on a server (backend service). To search for a query, the client generates trapdoors that disclose some private information and enables the server to search the encrypted data. Servers are often capable of satisfying queries without client-side support and the interaction between both parties is reduced to sending a query and retrieving the results.

One of the main goals of searchable encryption schemes is to provide efficient searches. To improve query latency, the server stores the database data in specialized data structures (indexes). Some constructions are static and build the index on the client side with an initialization phase. Once the index is stored on the server it can not be updated. Other schemes are dynamic and can insert, update and delete documents dynamically. One of the drawbacks of grouping related data in a data structure and using trapdoors to enable efficient server-side computation is the information disclosed. The adversary is generally static with access to all of the data and queries but can either be semi-honest or active. We mainly consider SSE schemes that are concerned with protecting the data confidentiality but the client is willing to disclose some partial information, a controlled leakage, in exchange of improved performance. As such, the security definitions of SSE schemes are parameterized by leakage functions that formally define the exact information revealed to the adversary. Overall, the leakage functions frequently followed in the state-of-the-art are summarized in the following list, with each function disclosing increasingly more information:

Index Information: One common leakage of SSE schemes is the correlation between the documents and the keywords which is preserved by the index stored in the server. For instance, a construction may disclose the size of the database, the number of documents, number of keywords and how many documents match a keyword.

Search Patterns: The leakage associated with queries is captured by the search patterns, i.e.: the set of tokens used by the client to search the for keywords in the untrusted server, index entries accessed by the server and the order in which an index is accessed during a query search. If the adversary can observe that two identical trapdoors or the same index entries are accessed for two distinct queries, then it can use statistical analysis to determine crucial information about the database keywords [KPT19, LMP18].

Backward Privacy: SSE schemes that support dynamic insertion and deletion of new documents on the index data structure can disclose the interleaving of these operations. A SSE scheme is said to be backward private if it does not disclose when keyword/document pairs are added and deleted, or which deletion cancels a prior insertion operation [BMO17].

Forward Privacy: Dynamic SSE schemes are said to be forward private if an update operation does not disclose the relation between a new document and the keywords stored on the index. [BMO17].

Access Pattern: The set of resulting records of a query define the access pattern leakage. The volume leakage is also associated to the access pattern and is defined as the size of the result set. This leakage as shown to be sufficient to reconstruct plaintext data on encrypted databases [GLMP18, GLMP19].

One of the first SSE scheme ever proposed (*Song et al.* [DWP00]) did not use an index but instead tagged every keyword in the database with a deterministic ciphertext. This approach only supports *exact match queries*, i.e.: search the documents that contain a single keyword. To evaluate a query, the client starts by generating deterministic a trapdoor from an input keyword. The trapdoor is sent to the server which scans over all of the keywords and documents stored in the database. For every trapdoor that matches a deterministic tag, the document identifier is returned to the client. This solution has limited performance with a linear time search over all keywords in all documents. Furthermore, it also has limited query expressiveness. Subsequent work has improved the performance of exact match queries considerable and expanded the set of supported queries to include boolean and range queries.

Exact Match Queries. Forward indexes provided the first solution to improve the asymptotic performance of exact math search queries. In these data structures, the index entry is an encrypted document identifier pointing to filter. The filter enables the server to test in constant time if a document contains a keyword given an input trapdoor. As such, the server evaluates queries by scanning the entire index and returning the documents containing the input keyword. The construction proposed by *Eu-Jin Goh* [Goh03] uses one bloom filter per document to test the membership of a keyword while the construction proposed by *Chang et al.* [CM05] uses a custom bit array data structure. Both constructions support dynamic indexes, but the latter construction assumes a fixed size set of keywords. However, the search time of forward indexes is far from optimal as it is linear to the number of documents.

The performance of search queries is improved even further by using inverted indexes. In an inverted index, each entry is an encrypted trapdoor that points to one or multiple documents containing a keyword. Constructions that use this index reduce the search space to the set of documents containing a keyword, resulting in sublinear search time. The first construction using

this approach was proposed by *Curtomla et al.* [CGKO11] and consists of an index where each entry is a keyword pointing to a list of documents containing the keyword. The documents as well as the keywords are encrypted, and a query is only satisfied if the server receives a trapdoor for the head of a specific list. This scheme is however limited to a static set of documents and cannot update the index by adding or removing documents. Subsequent work has proposed novel construction with support for dynamic operations [KPR12], improved performance of update queries by using a secondary index as a cache and lower search pattern leakages by leveraging ORAM constructions [SPS14]. The work of *Cash et al.* [CJJ+14] simplified existing constructions and presents a practical approach to process large datasets with parallel operations.

Extended Queries. SSE queries that only support exact match queries are essential but have a narrow applicability. Applications want to offload as much computation as possible to the database. Additionally, applications expect databases to support at least boolean queries that filter a set of documents by composing conjunctions and disjunctions of keywords. A naive approach to tackle this problem is to reduce a boolean query to multiple independent single keyword queries and aggregate the results on the client. This is not only inefficient but also leaks more information than necessary. *Cash et al.* proposed a more efficient solution with a query search time proportional to the least frequent keyword, i.e., the keyword with the least number of matching documents. This solution leverages two data structures, an exact match query SSE scheme which maps keywords to documents and a set data structure similar to a forward index. To search for a query, the client generates trapdoors for every keyword but generates a tag for the least frequent keyword. The server uses the tag in the the inverted index to obtain a list of documents. The resulting list is pruned by matching the document identifiers with the keyword tokens in the set data structure. However, this solution assumes a static index and is limited to conjunctive boolean queries. This work is extended and improved by *Faber et al.* to support queries that select a subset of documents between two keywords with a lexicographic order (range queries), queries that select documents based on a substring of a keywords and queries with a string combined with wildcard charactres, i.e., arbitrary characters. *Kamara et al.* [KM17] proposed an novel construction with a narrower set of queries but was the first construction to support disjunctive boolean queries and dynamic updates. The Recent work of *Bernardo et al.* [FPO+19] explored IEE technology and proposed a novel scheme that supports conjunctive and disjunctive boolean queries with forward and backward privacy [FPO+19].

3.4 General-purpose encrypted databases

Completely moving a production database to an untrusted third-party service while safeguarding the users confidentiality requires more than just SSE schemes. Relational Database Management Systems (RDBS) process rich queries with relational binary operators such as projections,

selections, joins and aggregations that slice, transform and otherwise process data into valuable information. Furthermore, the data types are not limited to keywords and also include numerical data as well as geographical and temporal data. Even key-value NoSQL databases that exchange the relational model for a stricter, specialized data model still support range queries, arithmetic operators and even aggregations. To preserve the same functionality, keep the server-side data encrypted and outsource as much computation as possible to the backend, novel solutions in the state-of-the-art coalesce a multitude of cryptographic techniques in a single system. Included in these techniques are Property Preserving Encryption (PPE) schemes, multiparty protocols and trusted hardware (IEE). The majority of these systems assume a semi-honest, static adversary.

Property-preserving databases. Conceptually, the idea of property-preserving databases is to encrypt plaintext data with multiple encryption schemes, each providing a different functionality. For instance, consider a database column with numerical attributes that is simultaneously queried to i) find the largest value and ii) sum two specific values. The column is encrypted with an OPE scheme to generate ciphertexts which can satisfy query i) and it is also encrypted with partial homomorphic encryption to satisfy query ii). At runtime, the database engine choses which ciphertext to use according to the input query. CryptDB [PRZB11] was the first system to propose and apply this idea using adjustable query-based encryption. The database client encrypts each table record item with onions of encryption, i.e., increasing stronger layers of encryptions stacked on top of each other. Database items are encrypted with a different secret key per item and the outermost onion layer is encrypted with a symmetric encryption scheme. The inner layers can be any combination of deterministic encryption, order preserving encryption, partial homomorphic encryption and even SSE schemes. With each query, the database adjusts the confidentiality of an item by peeling the outer layers as necessary. This adjustment process requires disclosing the decryption key for the server and permanently leaking the inner layers to the server. However, the secret key of the innermost layer that encrypts the plaintext is never disclosed to the server and is only used by client to decrypt query results.

CryptDB is one of the most influential systems in the literature and several solution have since improved on its contributions. Monomi [TKMZ13] builds on CryptDB and focus on analytical workloads. Its main contribution is a hybrid system with a split client/server query execution model. In this model, a query is optimized to execute all of the relational operators on the server side modulus the operators that cannot be efficiently computed over ciphertexts. For the latter operators the encrypted data is forwarded back to the client, decrypted and processed. Additionally, this system also fine tunes the cryptographic schemes by choosing more space efficient constructions to lower the overall storage and bandwidth usage. One common pitfall of both CryptDB and Monomi is the statistical information disclosed by queries that use the DET and OPE encryption schemes. *Naveed et al.* have showed that in specific application contexts (e.g.: medical field) this leakage is susceptible to frequency analysis attacks [NKW15] that can

successfully reconstruct plaintext values.

To mitigate this leakage some systems trade-off storage and performance overhead for security. *Antonis et al.* [MR92] proposed Seabed, a system that addresses frequency analysis attacks for the particular case of query aggregations. The system disperses the values of an attribute through multiple columns and also adds dummy values to the inserted records. The goal of these two mechanisms is to manipulate the frequency histogram of a column to have a uniform distribution. However, it requires a prior knowledge of the database queries as well as every possible combination of column aggregation. Additionally, the dummy values have a non-negligible storage overhead. The work of *Timon et al.* [TH20] extends the previous contribution and proposes SAGMA, a system that supports filter operators in aggregation queries. Conceptually, SAGMA transforms database tables in matrices. The values stored in the matrix are encrypted with a partial homomorphic scheme that provides semantic security and queries are evaluated as linear algebra operations that naturally touch every matrix element (e.g.: matrix multiplication), thus hiding the frequency distribution. However, this transformation requires assigning numerical value to non-numerical attributes and keeping this relation in an SSE scheme.

The main ideas behind PPE protected databases have also been applied to NoSQL databases. *Xingliang et al.* [YWW+16] were the first to consider the problem of data locality in distributed key-value databases. In these systems, data is partitioned horizontally in multiple regions and stored in a distributed file system. If there is a mismatch between the storage node holding a region and the database node that processes queries over a region, then an additional network request is necessary to transfer the data between the nodes. To mitigate this issue, the authors propose a framework with a novel data partitioning algorithm that uses a PRF to ensure that each region is consistently hashed to the correct storage and database node. *Zhen et al.* [ZLP+17] address a different problem, the incompatibility between compression and encryption. NoSQL databases handle large amounts of data by compressing data to improve bandwidth usage and fit more information in the main memory. However, if data is encrypted then the compression algorithm cannot achieve high compression ratios. The authors proposed MiniCrypt, a system that joins multiple records together in individual packs. Each pack is assigned a unique identifier and its contents are compressed and encrypted. Both of these systems disclose statistical information susceptible to frequency analysis attacks. Arx [PBP19] is the first system proposed to address this issue in NoSQL databases by using SSE that ensure semantically security. However, this approach needs to re-encrypt part of the database after every query, resulting in a significant network bandwidth overhead.

Multiparty databases. Whereas PPE cryptographically protected databases rely on multiple encryption schemes, multiparty protocols can be used to compute any function using a single scheme. However, the practical applicability of these protocols has been lacking and mostly

focused on the the classical setting where independent parties want to compute a common function. In this setting, Sharemind [BLW08] provides one of the most stable, flexible and industry proven frameworks. But in the setting of CPD, *Wai et al.* [WKC+14] were the first to explore these approach by proposing a two-party SQL database. In this system, one of the parties is the database client and the other is the database server. Data is encrypted with a secret sharing scheme in such a way that the majority of the data is stored on the server, but a few secrets have to be kept on the client side to prevent an adversary from reconstructing the plaintext values. Queries are processed by novel two-party composable protocols that support a subset of SQL operators: selection, projection, joins, addition and multiplications. Besides the client-side information these protocols also require the client to take part on the protocol evaluation and don't completely offload the computation to the server.

Trusted hardware databases. Trusted hardware modules, an IEE in Section 2.3.3, can significantly enhance cryptographic schemes by processing plaintexts on the server side. At the core of trusted hardware is the ability to allocate secure memory regions that are inaccessible by any external environment, including the operating system, hypervisors and even hardware directly connected to the system buses. Applications can load sensitive data as well as arbitrary programs on to a secure memory region only through a public interface protected by the hardware. All of unauthorized accesses to read or modify the contents inside the secure memory regions are refused. Programs loaded on to a trusted hardware can generate a proof of integrity. This proof enables client applications to verify that programs have been correctly loaded on to a trusted hardware and that the hardware is genuine, i.e., it is not a third-party emulating a trusted hardware. With these security guarantees, software running inside the secure memory regions can process data as plaintexts. However, all of the sensitive data outside the secure memory regions still needs to be encrypted. Including the data that is loaded on to the secure memory regions. This data can only be decrypted once inside the trusted hardware. An adversary can observe all requests made from the external environment on to the secure memory regions as well as any request on the opposite direction. Current trusted hardware has some technical limitations, the size of the secure memory regions is small and the execution of software inside these regions has some performance overhead. Additionally, these systems are vulnerable to side-channel attacks and there are a few security exploits that have successfully extracted secret keys which compromise the overall system security [BMW+18, BSN+19].

Trusted hardware provides, at least theoretically, a simple solution for CPD. However, a database engine cannot simply be deployed inside an IEE due to memory constrains, performance overhead, technical and security limitations. TrustDB [BS11] overcame these restrictions in relational databases with a split-execution approach that divides data in public and private attributes. Public attributes are stored as plaintext and can be processed by the database engine without any modification. Private attributes are encrypted with an IND-CPA scheme and any operations over these attributes is executed inside an IEE. This solution is limited to small fine-

grain relational operators and only protects a fraction of a database engine. EnclaveDB [PVC18] widens the set of database components protected by an IEE to the query engine and the transaction manager. Additionally, it also ensures data integrity even in the presence of an active adversary. However, the protected database is in-memory system that assumes a fixed scheme and queries that can not be updated after an initial setup phase. Furthermore, even though trusted hardware such as Intel SGX provides transparent encryption of the main memory used by an IEE, the access patterns are still disclosed to the adversary. *Ankur et al.* [DLP+20] and *Wenting et al.* [ZDB+17] have made some progresses to address this issue in analytical and federated databases respectively. These solutions leverage existing oblivious algorithms as well as propose novel specialized operators to join, filter and aggregate data that does not disclose the access patterns, albeit at a significant performance overhead.

3.5 Summary

The sum total of existing contributions in secure outsource computation amounts to offload increasingly more computation to an untrusted server in exchange for either performance or security. The least intrusive option is to keep the database engine in a trusted site and swap the underlying file system for a remote encrypted file system that stores data in the untrusted site. Currently, there are two divergent approaches of encrypted file systems based on the internal architecture, kernel space and user space systems. Both are viable solutions, with kernel space systems having traditionally a performance advantage due to lower number of context switches. However, hardware improvements and optimized frameworks such as Fuse have lessened the performance overhead and placed users space systems as a feasible approach to develop novel efficient cryptographic file systems that can provide greater extensibility and customization than kernel space systems.

Searchable encryption schemes provide a solution in between a client side only system and a general server-side CPD. Research in SSE schemes is centered on data structures, forward indexes or backward indexes, that support efficient exact match queries and even searches with boolean formulas. However, these schemes have an associated leakage that if not thoroughly considered can leave the database vulnerable to malicious attacks. Moreover, searching data is only one operation of relational and NoSQL system. For this reason, PPE databases have also leveraged additional encryption schemes such as OPE and partial homomorphic encryption to process complex queries on the server side without any client-side support. There are alternative solutions that do not require to use a multitude of conflicting cryptographic schemes, each one possible introducing security exploits. Secure multiparty protocols are one alternative that has been mostly left unexplored in the context of encrypted databases despite providing a unified processing framework without functional limitations. The other alternative is trusted hardware solutions where existing research focus on finding best vertical division of a database

architecture that simultaneously lowers the performance overhead of executing queries in a protected environment and maximizes the security guarantees.

Chapter 4

Multi-layer encrypted file system for private databases

In this chapter we present our first main contribution of the thesis, SafeFS, a user-space multi-layer encrypted file system. SafeFS is new file system that if placed between a database system and the physical storage can manage the data plane to provide security, performance and availability properties. This system improves on the state-of-the-art with its modular architecture that provides a straightforward approach to stack and change logical layers in user-space. The layers can be adjusted to fit the client application requirements and workload. We showed through an extensive experimental evaluation that our approach has a minimal performance overhead in classical storage workloads as well as transactional and analytical database workloads.

4.1 Introduction

Offloading the storage of a database system to an untrusted third-party service raises interoperability and privacy issues. Market leading cloud providers (e.g.: Google [Goo], Amazon [Ama] and Azure [Micb]) have several specialized storage products such as elastic storage, cross region storage, block-oriented storage and archive storage. Each one of these products address a specific requirement such as on-demand scalable storage for large amounts of data, high-throughput storage for transactional workloads and even secondary storage for backups or archives that are sparsely accessed. However, the interface of these solutions is heterogeneous and often incompatible with database systems. Furthermore, simply storing the data on third-party service without encrypting it can result in a confidentiality breach.

The first problem can be partially solved by providing well-known and extensively used abstractions on top of third-party interfaces [Wal95]. One of the most widespread and high-level abstractions offered atop storage systems is the POSIX I/O file system interface [Wal95]. The

practicality and ease of use of such abstraction has spurred the development of a plethora of different file system solutions offering a variety of compromises between I/O performance optimization, availability, consistency, resource consumption, security, and privacy guarantees for stored data [UAA+10, BMO+14]. Additionally, developers can leverage the Fuse framework to implement a POSIX-like file system on top of a multitude of local and remote storage systems in a fairly straightforward manner. Nevertheless, each file system implementation is different and specifically designed for certain use cases or workloads. Choosing which implementations to use and combining them in order to take advantage of their respective strong features is far from trivial [ZN00].

The second problem, securely outsourcing sensitive data, has also been the subject of intensive work. There are several remote encrypted file system implementations providing privacy preserving mechanisms [Hal10, enc]. However, similarly to storage systems, a single approach does not address the specific privacy needs of every application or system. Some require higher levels of data privacy while others target performance at the price of lower privacy guarantees. Furthermore, these approaches lack a clear separation between privacy preserving mechanisms and the file system implementation itself. This prevents a seamless combination of different privacy preserving mechanisms with other file system properties (*e.g.*, caching, compression, replication, etc).

In this chapter we tackle both challenges simultaneously. Inspired by Software-Defined Storage (SDS) design principles [TBO+13, AAB+14], we introduce SafeFS, a novel approach to encrypted user-space file systems. We advance the state of the art in two important ways by providing *two-dimensional modularity* and *extensible security mechanisms*. First, SafeFS two-dimensional modular design can combine implementations of specialized storage layers for security, replication, coding, compression and deduplication, while at the same time allowing each layer to be individually configurable through plug-in software drivers. SafeFS layers can then be stacked in any desired order to address different application needs. The design of SafeFS avoids usual pitfalls such as the need for global knowledge across layers. For instance, for size-modifying layer implementations (*e.g.*, encryption with padding, compression, deduplication), SafeFS does not require a cross-layer metadata manager to receive, process, or redirect requests across layers [ZABN01]. Second, SafeFS design allows us to easily combine any Fuse-based file system implementation with several cryptographic techniques and, at the same time, to leverage both centralized [webf, webe] and distributed storage backends [BMO+14]. For example, it is straightforward to integrate an existing Fuse-based file system with secret sharing on top of distributed storage backends using SafeFS simply by adapting the system APIs.

To show the practicality and effectiveness of our approach, we implemented a full prototype of SafeFS that, as in recent proposals [BGG+09, VTZ17], resorts to the Fuse framework. With a thorough experimental evaluation, we compare several unique configurations of SafeFS, each combining different privacy-preserving techniques and cryptographic primitives. We evaluate the

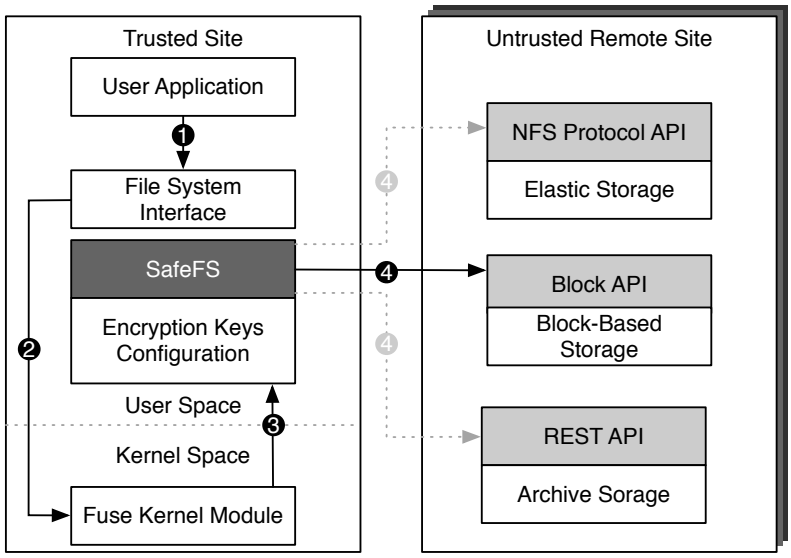


Figure 4.1: Conceptual model of outsourcing storage to an untrusted remote site.

performance by resorting to state-of-the-art benchmarks, including a file system benchmark, a transactional database benchmark as well as web-based database benchmarks. The remainder of this chapter is organized as follows. In Section 4.2.2 we present a conceptual definition of the problem addressed in this chapter and illustrate the design goals of SafeFS. The following section 4.3 details SafeFS's architecture and the approach used to solve the aforementioned problems. The implementation details are given in Section 4.4. Section 4.5 presents our extensive evaluation of the SafeFS prototype, before providing a discussion of the overall contribution in Section 4.6.

4.2 Problem definition

Moving the database storage to a third-party service can be done at two levels, either at the database engine level or at the file system level. Modifying the database engine is an intrusive approach that requires in depth knowledge and can not be seamless applied to different systems. The latter approach is non-intrusive and cross compatible with multiple systems as file systems use a common POSIX I/O file system interface. SafeFS is applied at the second level to create a solution that is orthogonal to any application.

To capture the problem of storage outsourcing addressed by SafeFS we depict in Figure 4.1 a general model that contains the most relevant components and their interactions. The model is divided in two independent sites, the *trusted site* and an *untrusted site*. The *untrusted site* abstracts any cloud provider with a storage solution that is accessible through a public interface, either an NFS protocol, a REST protocol or any other proprietary interface. The *trusted site* can use the public interface to store, access, delete and modify data at will without any size

constrains or limitation on the number of requests. However, the *untrusted site* does not provide any computational services. The *trusted site* can use more than a single *untrusted site* and use a different interface in each one. Internally, the *trusted site* is split between the *user space* and *kernel space*. On the *user space* reside the *user application* (e.g.: Database system), the *file system* interface and SafeFS which manages some encryption keys and custom configuration. The user application as well as the *file system* interface are left unmodified. On the *kernel space* we only consider the *fuse kernel module* that intercepts file system request and relays them to SafeFS.

We provide an overview of the interaction between the components which is further detailed in Section 4.3.1. Requests start on the *user application* to update or access a file in the underlying file system (Figure 4.1-❶). In this interaction, data is always accessed or updated as plaintext. From the application perspective, the file system behaves as a native file system. However, when the request is sent to the *kernel space* it is eventually intercepted by the *fuse kernel module* (Figure 4.1-❷). If the request is made to a directory mounted by SafeFS, then the *fuse kernel module* forwards the request (Figure 4.1-❸). SafeFS can process incoming requests in different ways depending on the configuration of layers. If SafeFS is configured with an encryption layer, then the data in the request is encrypted and forwarded to one or multiple *remote site* interfaces (Figure 4.1-❹). The reverse path is taken to return the result of the operation back to the *user application* and provide a consistent file system view. Throughout the remaining chapter we will focus only on presenting SafeFS architecture from the *trusted site* viewpoint.

4.2.1 Trust model

We assume the *untrusted site* is corrupted by an semi-honest adversary that observes every requests received. In fact, the adversary knows how many requests are made, what type of operations are made (e.g.: read or write), the access pattern of the requests (e.g.: sequential, random) and can view the data stored. All of this can be learned by the interaction between the *trusted site* and *untrusted site* (Figure 4.1-❹) which is not protected by a secure channel. However, everything that happens within the *trusted site* is outside of the adversary control. Furthermore, we assume the adversary does not try to compromise the integrity of the data or forge the authenticity of the requests. In fact, SafeFS can support this type of attacks by using authenticated encryption but the current version is focused only on ensuring data confidentiality.

4.2.2 Design goals

SafeFS is a framework for flexible, modular and extensible file system implementations built atop Fuse. Its design allows to stack independent layers, each with their own characteristics, and optimizations. These layers can then be integrated with existing Fuse-based file systems as well as re-stacked in different order. Each stacking configuration leads to file systems with different

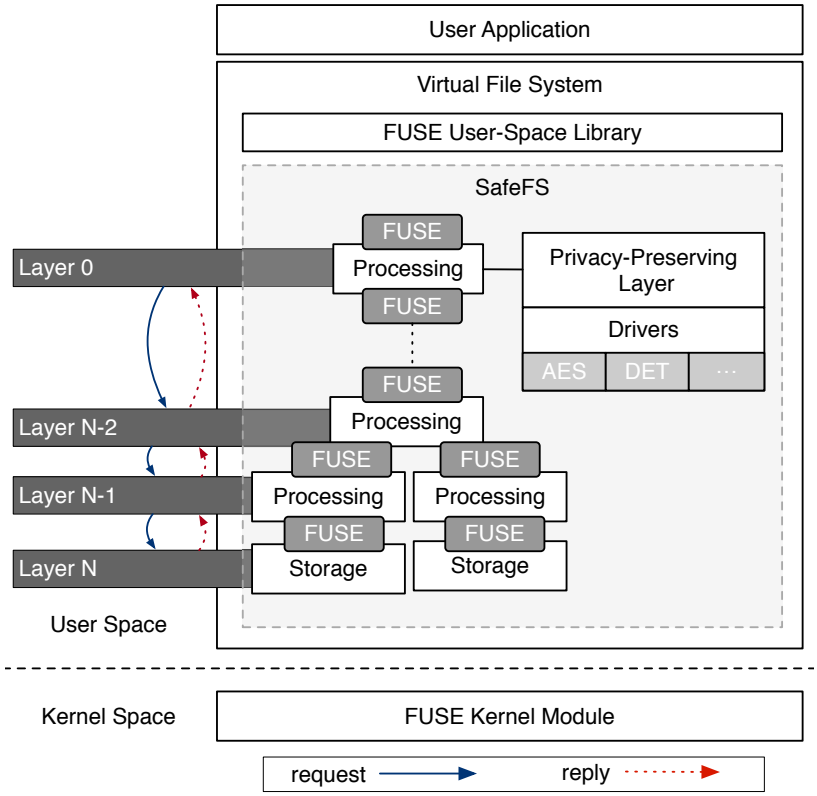


Figure 4.2: Architecture of SafeFS.

traits, suitable to different applications and workloads. Keeping this in mind, the four pillars of our design are:

- *Effectiveness.* SafeFS aims to reduce the cost of implementing new file systems by focusing on self-contained, stackable, and reusable file system layers.
- *Compatibility.* SafeFS allows us to integrate and embed existing Fuse-based file systems as individual layers.
- *Flexibility.* SafeFS can be configured to fit the stack of layers to the applications requirements.
- *User-friendliness.* From a client application perspective, SafeFS is transparent and usable as any other Fuse file system.

4.3 Architecture

Figure 4.2 depicts the architecture of SafeFS. The system exposes a POSIX-compliant file system interface to the client applications. Similar to other Fuse systems, all file system related

operations (*e.g.*, `open`, `read`, `write`, `seek`, `flush`, `close`, `mkdir`, etc.) are intercepted by the Linux Fuse kernel module and forwarded to SafeFS by the Fuse user-space library. Each operation is then processed by a stack of layers, each with a specific task. The combined result of these tasks represents a file system implementation.

We identify two types of SafeFS layers serving different purposes. Upon receiving a request, *processing* layers manipulate or transform file data and/or metadata and forward the request to the next layers. Conversely, *storage* layers persist file data and metadata in designated storage backends, such as local disks, network storage, or cloud-based storage services. All layers expose an interface identical to the one provided by the Fuse library API, which allows them to be stacked in any order. Requests are then orderly passed through all the layers such that each layer only receives requests from the layer immediately on top of it and only issues requests to the layer immediately below. Layers in the bottom level must be of storage type, in order to provide a functional and persistent file system.

This stacking flexibility is key to efficiently reuse layer implementations and adapt to different workloads. For example, using compression before replicating data across several storage backends may be acceptable for archival-like workloads. In such settings, decompressing data before reading it does not represent a performance impairment. On the other hand, for high-throughput workloads it is more convenient to only apply compression on a subset of the replicated backends. This subset will ensure that data is stored in a space-efficient fashion and is replicated to tolerate catastrophic failures, while the other subset will ensure that stored data is uncompressed and readily available. In these scenarios, one storage stack would use a compression layer before a replication layer, while a second storage stack would put the compression layer after the replication and only for a subset of storage backends. Layers must be stacked wisely and not all combinations are efficient. An obviously bad design choice would be to stack a randomized privacy layer (*e.g.*, standard AES cypher) before a compression layer (*e.g.*, gzip): by doing so, the efficiency of the compression layer would be highly affected since information produced by the above layer (the randomized encryption) should be indistinguishable from random content.

Finally, the SafeFS architecture allows us to embed distributed layers as intermediate layers. This is depicted in Figure 4.2, where layer $N-1$ (*e.g.*, a replication layer) stores data into two different sub-layers N. SafeFS supports redirection of operations toward multiple layers, while at the same time maintaining these layers agnostic from the layer above that transmits the requests.

4.3.1 A day in the life of a write

To illustrate the I/O flow inside a SafeFS stack, we consider a `write` operation issued by the client application to the virtual file system (`read` operations are handled similarly). Each request made to the virtual file system is handled by the Fuse kernel module (Figure 4.3-❶)

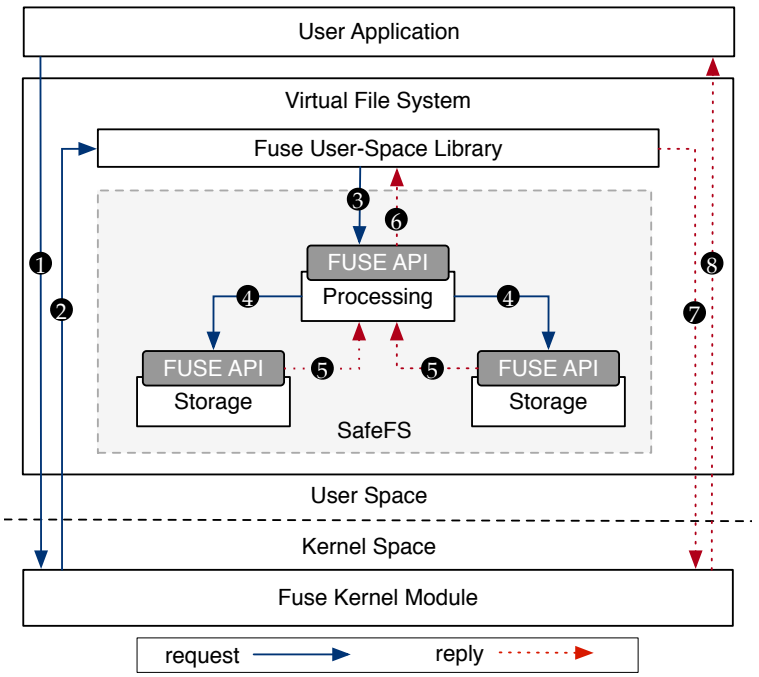


Figure 4.3: Execution flow of a `write` request.

that immediately forwards is to the user-space library (Figure 4.3-❷). At this point the request reaches the topmost layer of the stack (Figure 4.3-❸), called *Layer 0*. After processing the request according to its specific implementation, each layer issues a write operation to the following layer. For example, a privacy-preserving layer responsible for encrypting data will take the input data, encrypt it according to its configuration, and emit a new `write` operation with the ciphertext to the underlying layer. This process is repeated, according to each layer implementation, until the operation reaches a *storage* layer, where the data is persisted into a storage medium (Figure 4.3-❹). The reply request stating whether the operation was successfully executed or not takes the reverse path and is propagated first to *Layer 0* (Figure 4.3-❺), and eventually backward up to the application (Figure 4.3-❽).

When using distributed layers (*e.g.*, with replication), `write` operations are issued to multiple sublayers or storage backends. These distributed layers can break some of the assumptions made by the applications. For instance, `rename` and `sync` operations must be atomic. To ensure correct semantics of the operations, a distributed layer should contain a synchronization mechanism that ensures that an operation is only committed if successful in every backend. Otherwise, the operation fails, and the file state must not be changed. A possible solution would be a block cache that stores blocks before any operation is applied.

We have discussed so far how layers modify data from read and write operations. The behavior for layers that modify the attributes and permissions of files and folders is similar. For instance, a layer providing access control to files shared among several users will add this behavior to the specific Fuse calls that read and modify the files. This design paves the way

for layer reuse and for interesting stacking configurations. Individual layers do not need to implement the totality of the Fuse API: if a layer only manipulate files, it only needs to wrap the Fuse operations that operate over files. Fuse operations over folders can be ignored and passed directly to the next layer without any additional processing. Layers can support the full Fuse API or a restricted subset, and this allows for a highly focused layer development cycle.

4.3.2 Layer integration

Besides the standard Fuse API, each SafeFS layer implements two more functions. First, the *init* function initializes metadata, loads configurations, and specifies the following layer(s) in the specific SafeFS stack. Second, the *clean* function frees the resources possibly allocated by the layer. The integration of existing Fuse-based implementations in the form of a SafeFS layer is straightforward. Once the *init* and *clean* are implemented, a developer simply needs to link its code against the SafeFS library instead of the default Fuse. Additionally, for a layer to be stacked, delegation calls are required to forward requests to the layers below or above. The order in which layers are stacked is flexible and is declared via a configuration file. Finally, SafeFS supports layers that modify the size of data being processed (*e.g.*, compression, padded encryption) without requiring any global index or cross-layer metadata. This is an advantage over previous work [WMZ03b], further discussed with concrete examples in Section 4.4.

4.3.3 Driver mechanism

Some of the privacy-preserving layers must be configured with respect to the specific performance and security requirements of the application. However, these configurations do not change the execution flow of the messages. From an architectural perspective, using a DES cipher or an AES cipher is strictly equivalent.

With this observation in mind, we further improved the SafeFS modularity by introducing the notion of *driver*. Each layer can load a number of drivers by respecting a minimal SafeFS driver API. Such API may change according to the layer specialization and characteristics, as further discussed in the next section. Drivers are loaded according to a configuration file at file system's mount time. Moreover, it is possible to change a driver without recompiling the file system, re-implement layers, or to load new layers. Naturally, this is possible provided that the new configuration does not break compatibility with the previous one. For instance, introducing different cryptographic techniques will prevent the file system from reading previous data.

Consider the architecture depicted in Figure 4.2, with a privacy-preserving layer having two drivers, one for symmetric encryption via AES and another for asymmetric encryption with RSA. The driver API of the layer consists of two basic operations: `encode` and `decode`. In this scenario, the cryptographic algorithms are wrapped behind the two operations. When a `write` request is

intercepted, SafeFS calls `encode` on the loaded driver and the specific cryptographic algorithm is executed. Similarly, when a `read` operation is intercepted, the corresponding `decode` function is called to decipher the data. In order to change the driver, it is sufficient to unmount the file system, modify the configuration file, and remount the SafeFS partition.

The driver mechanism can be exploited by layers with diverse goals, such as those targeting compression, replication, or caching. In the next section we discuss the current implementation of SafeFS and illustrate its driver mechanism in further details.

4.4 Implementation

We have implemented a complete prototype of SafeFS in the C programming language. Currently, it consists of less than 4,200 lines of code (LOC), including headers and the modules to parse and load the configuration files. Configuration files are used to describe what layers and drivers are used, their initialization parameters, and their stacking order. The code required to implement a layer is also remarkably concise. For example, our cryptography-oriented layer only consists of 580 LOC. SafeFS requires a Linux kernel that natively supports Fuse (v2.6.14). To evaluate the benefits and drawbacks of different layering combinations, we implemented three unique SafeFS layers, as depicted in Figure 4.4. These layers are respectively concerned with data size normalization (*granularity-oriented*), enforcing data privacy policies (*privacy-preserving*) and data persistence (*multiple backend*). Since they are used to evaluate SafeFS, we detail them in the remainder of this section.

4.4.1 Granularity-oriented layer

It is important to be able to stack layers that operate on data at different granularity levels, *e.g.*, with different block sizes. For example, one might need to stack a layer that reports dynamic sizes for file `write` and `read` operations over a layer that only works with fixed-sized blocks ([WMZ03a, enc]).

As a more concrete example, the Fuse user-space library reports file `write` and `read` operations with dynamic sizes. Yet, many cryptographic algorithms only work with fixed-size block granularity and hence require a *block size translation* mechanism. Such translation is provided by the *granularity-oriented* layer. This layer opens the way to exploit block-based encryption, instead of whole-file encryption, which is more efficient for many workloads where requests are made only for small portions of the files. For instance, if only 3 bytes of a file are being read and the block size is 4KB, then only 4KB must be deciphered while a whole-file approach could require the entire file to be deciphered only to recover those same 3 bytes of data.

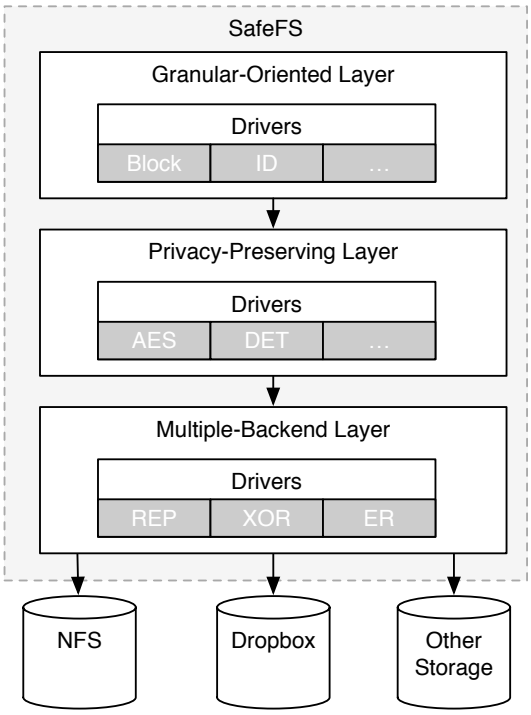


Figure 4.4: SafeFS: chain of layers and available drivers.

In more details, the translation layer creates a logical abstraction on top of the actual data size being read, written, or modified. This is achieved by processing data `write` and `read` requests from the upper layer and manipulating the offsets and data sizes to be read/written from underneath layers. The manipulation of the offsets and sizes is done using two functions: `align_read` and `align_write`. The drivers of the layer must implement both function calls to define distinct logical views for read (for `align_read`) and write (for `align_write`) operations. Operations on directories or file attributes are redirected to adjacent layers pristine.

Our prototype implements two drivers for the translation layer: a *block* and an *identity* driver (ID). The *block* driver creates a logical block representation for both file write and read requests, which will be used transparently by the following layers. This block abstraction is fundamental for layers whose processing or storage techniques rely on block-based requests (*e.g.*, block-based encryption, de-duplication, etc.). Block size is configured on driver initialization. On the other hand, the *identity* driver does not change the offset or the buffer size of the bytes read or written. We use this driver as a baseline to understand the overhead of our block-oriented approach and the layer itself.

4.4.2 Privacy-preserving layer

The goal of this layer is to protect sensitive information in a transparent way for applications and other layers of SafeFS. As explained in Section 4.3.3, file data being written or read is intercepted by the layer and then ciphered (`encode`) or deciphered (`decode`). We support three drivers: standard AES encryption (*AES*), deterministic encryption (*DET*), and *Identity* (ID).

The *AES* driver leverages the OpenSSL's own AES-128 block cipher in *CBC* mode [webd]. Both key and initialization vector (`IV`) size of the AES cipher are parameters defined during the initialization of the driver. Our design follows a block-based approach for encrypting and decrypting data. Hence, the *block* driver of the *granularity-oriented* layer is crucial to transparently ensure that each `encode` and `decode` call issued by the *AES* driver receives the totality of the bytes for a given block. Each block has a random IV associated, generated in the *encode* function, that is stored as extra padding to the cipher text. The `IV` is important for decoding the ciphertext and returning the original plaintext but keeping it public after encryption does not impact the security of the system [BR05].

The *Det* driver protects the plaintext with a block-based deterministic cipher (This cipher does not need a new random `IV` for each encoded block and is hence faster than randomized encryption. Despite compression algorithms being more efficient in plaintext, this driver helps detect data redundancy over ciphertext, otherwise impossible to find with a standard randomized encryption scheme.

Both drivers resort to padding (16 bytes from the AES padding plus 16 bytes for storing the `IV`). For example, a 4KB block requires 4,128 bytes of storage. Manipulating block sizes must be done consistently across file system operations. Every size modifying layer must keep track of the real file size and the modified file size, so no assumption is broken for the upper and lower layers. For instance, if a layer adds padding data, it only reports the original file size without the extra padding to the previous stack layer.

Finally, we implemented an *Identity* driver, which does not modify the content of intercepted file operations and is used as an evaluation baseline, similarly to the *granularity-oriented* Identity driver. We note that drivers for other encryption schemes (*e.g.*, DES, Blowfish, or RSA) could be implemented similarly.

4.4.3 Multiple backend layer

The storage layers directly deal with persisting data into storage backends. In practice, these backends are mapped to unique storage resources available on the nodes (machines) where SafeFS is deployed. The number of storage backends is a system parameter. They may correspond to local hard drives or remote storage backends, such as NFS servers accessible via local mount points. The drivers for this layer follow the same implementation pattern described

previously, namely via `encode` and `decode` functions. The encode method, upon a write request, is responsible for generating a set of data pieces to be written in each storage backend from the original request. The decode implementation must be able to recover the original data from a set of data pieces retrieved from the storage backends.

Our evaluation considers three drivers: replication (*REP*), secret sharing (*XOR*), and erasure coding (*Erasure*). The *Rep* driver fully replicates the content and attributes of files and folders to all the specified storage backends. Thus, if one of the backends fails, data is still recoverable from the others. The *XOR* driver uses a bitwise additive secret sharing to protect files. The driver creates a secure block (secret) by applying the $bitwiseexclusiveor$ to a file block and a random generated block. This operation can be applied multiple times using the previous new secret as input, thus generating multiple secrets. The original block can be discarded and the secrets safely stored across several storage backends [GMW87]. The content of the original files can only be reconstructed by accessing the corresponding parts stored across the distinct storage backends. Finally, the *Erasure* driver uses erasure codes such as Reed-Solomon codes [Wic94] to provide reliability similar to replication but at a lower storage overhead. This driver increases data redundancy by generating additional parity blocks from the original data. Thereafter, a subset of the blocks is sufficient to reconstruct the original data. The generated blocks are stored on distinct backends, thus tolerating the unavailability of some of the backends without any data loss. As erasure codes modify the size of data being processed, this driver resorts to a metadata index that tracks the offsets and sizes of stored blocks on a per-file basis. The index allows containing the size-changing behavior of erasure-codes within the layer, thus not affecting any other layer.

4.4.4 Configuration stacks

The above layers can be configured and stacked to form different setups. Each setup offers trade-offs between security, performance, and reliability. The simplest SafeFS deployable stack consists of the *multiple backend* layer plus the *Rep* driver with a replication factor of 1 (file operations issued to a single location). This configuration offers the same guarantees of a typical Fuse loopback file system.

Increasing the complexity of the layer stack leads to richer functionalities. By increasing the replication factor and the number of storage backends for the simplest stack, we obtain a file system that tolerates disk failure and file corruption. Similarly, replacing the *Rep* with the *Erasure* driver, one may achieve a file system with increased robustness and reduced storage overhead. However, erasure coding techniques only work in block-oriented settings thus requiring the addition of the *granularity-oriented* layer to the stack.

When data privacy guarantees are required, one simply needs to include the *privacy-aware* layer into the stack. Due to the block-based nature of the *Rep* and *Erasure* drivers, the *granularity-*

oriented is also required. However, note that when *AES* and *Erasure* are combined, the file system stack only requires a single *block* oriented layer. This layer provides a logical block view for requests passed to the *privacy-aware* layer. These requests are then automatically passed as blocks to the *multiple backend* layer.

Using the *XOR* driver provides an interesting privacy-aware solution, since trust is split on several storage domains. This driver exploits a bitwise technique not dependent on previous bytes to protect information, thus it does not require a block-based view as *privacy-aware* drivers.

4.5 Evaluation

This section presents our comparative evaluation of the SafeFS prototype. First, in Section 4.5.1 we present the third-party file systems against which we compare SafeFS. Then, in Section 4.5.2, we describe the selected SafeFS stack configurations and their trade-offs. Section 4.5.3 presents the evaluation methodology and the benchmark tools. Finally, Section 4.5.4 and Section 4.5.5 focuses on the evaluation results.

4.5.1 Third-party file systems

Since our SafeFS prototype focuses on encrypted file systems, we deployed and ran our suite of benchmarks on three well-known open-source file systems with encryption capabilities. More precisely, we evaluate SafeFS against the CryFS [cry] and EncFS [enc] user-space file systems. We further include eCryptfs [Hal10], a kernel-space file system available in the Linux mainstream kernel. We selected those for being widely used, freely available, adopted by the community, and offering different security trade-offs. We detail the characteristics of each system relevant for the evaluation:

CryFS (v0.9.6) is a Fuse-based encrypting file system that ensures data privacy and protects file sizes, metadata, and directory structure. It uses AES-GCM for encryption and is designed to integrate with cloud storage providers such as Dropbox.

EncFS (v1.7.4) is a cross-platform file system also built atop Fuse. This system has no notion of partitions or volumes. It encrypts every file stored on a given mounting point using AES with a 192-bit key. A checksum is stored with each file block to detect corruption or modification of stored data. In the default configuration, also used in our benchmarks, a single `IV` is used for every file, which increases encryption efficiency but decreases security.

eCryptfs (v1.0.4) includes advanced key management and policy features. All the required meta-data are stored in the file headers. Similar to SafeFS, it encrypts the content of a

mounted folder with a predefined encryption key using AES-128.

4.5.2 SafeFS configurations

		Granularity-Oriented		**Privacy-Preserving**			**Multiple-Backend**		
Groups	Stack	Block	ID	AES	DET	ID	REP	XOR	Erasure
Baseline	Fuse	×	×	×	×	×	√,1	×	×
	Identity	×	√	×	×	√	√,1	×	×
Privacy	*AES*	√	×	√	×	×	√,1	×	×
	DET	√	×	×	√	×	√,1	×	×
	XOR	×	×	×	×	×	×	√,3	×
Redundancy	*REP*	×	×	×	×	×	√,3	×	×
	Erasure	√	×	×	×	×	×	×	√,3

Table 4.1: The different SafeFS stacks deployed in the evaluation. Stacks are divided in three distinct groups: Baseline, Privacy, Redundancy. The table header holds the three SafeFS layers. Below each layer we show the respective drivers. For each stack, we indicate the active drivers (the √ symbol). Layers without any active drivers are not used in the stack. The indices for Multiple-Backend drivers indicate the number of storage backends used to write data.

Our benchmarks use a total of 7 different stack configurations (Table 4.1). Each exposes different performance trade-offs and allows us to evaluate the different features of SafeFS. The chosen stacks are divided in three groups: baseline, privacy, and redundancy.

The first group of configurations, as the name implies, serve as baseline file system implementations where there is no data processing. The Fuse stack is a file system loopback deployment without any SafeFS code. It simply writes the content of the virtual file system into a local directory. The *Identity* stack is an actual SafeFS stack where every layer uses the *identity* driver. It corresponds to a pass-through stack where the storage layer mimics the loopback behavior. These two stacks provide means to evaluate SafeFS framework overhead and individual layer overhead.

The privacy group is used to evaluate the modularity of SafeFS and measure trade-offs between performance and privacy guarantees of different privacy preserving techniques. In our experiments we used three distinct techniques. The `AES` stack and `DET` stacks correspond respectively to a standard and a deterministic encryption mechanism. The `AES` stack is expected to be less efficient than `DET` as it generates a different IV for each block. However, `DET` has the weakest security guarantee. The third stack, named `XOR`, considers a different trust model where no single storage location is trusted with the totality of the ciphered data. Data is stored across distinct storage backends in such a way that unless an attacker gains access simultaneously to all backends, it is impossible to recover any sensitive information about the original plaintext.

Finally, the two remaining stacks deal with data redundancy. The `REP` stack fully replicates

files into three storage backends. In our configuration, two out of three backends can fail, while still allowing the applications to recover the original data. The `Erasure` stack serves the same purpose but relies on erasure codes for redundancy instead of traditional replication. Data is split into 3 fragments (2 data + 1 parity) over 3 backends for a reduced storage overhead of 50%, with respect to replication. This erasure configuration supports the complete failure of one of the backends. These stacks provide an overview of the costs of two different redundancy mechanisms.

4.5.3 Methodology

Our evaluation is divided in two settings, a micro-benchmark setting and a macro-benchmark setting.

Micro-benchmark setting. The goal of the micro-benchmark setting is to compare the performance of SafeFS with the third-party systems in a local setting where all of the data storage is made on a single host, the trusted site. Furthermore, this setting only uses general benchmarks that directly measure the latency of different file system operations without the overhead of a real application. We conducted our micro-benchmark experimental evaluation using two commonly used benchmarking suits: `db_bench` and `filebench`. The `db_bench` benchmark is included in `LevelDB`, an embeddable key-value store [webc]. This benchmark runs a set of predefined operations on a local key-value store installed in a local directory. It reports performance metrics for each operation on the database. The `filebench` [webb] tool is a full-fledged framework to benchmark file systems. It emulates both real-world and custom workloads configured using a workload modeling language (WML). Its suite of tests includes simple micro-benchmarks as well as richer workloads that emulate mail- or web-server scenarios. We leverage and expand this suite throughout our experiments.

The experiments ran on virtual machines (VM) with 4 cores and 4GB of RAM. The KVM hypervisor exposes the physical CPU to the guest VM with the `host-passthrough` option [webg]. The VMs run on top of hard disk drives (HDD) and leverage the `virtio` module for better I/O performance. We deployed each file system implementation inside a Docker (v1.12.3) container with data volumes to bypass Docker's AUFS [weba] and hit near-native performance.

Macro-benchmark setting. The macro-benchmark aims to measure the overhead of a remote encrypted file system in a production database. In this setting we consider a distributed deployment with a database engine on a trusted site outsourcing storage to one or more remote backends in an untrusted site. More concretely, we use the PostgreSQL database management engine [Mac16]. This database is considered one of the most efficient and reliable database systems used in the industry due to its long-history of open-source development. For the remote

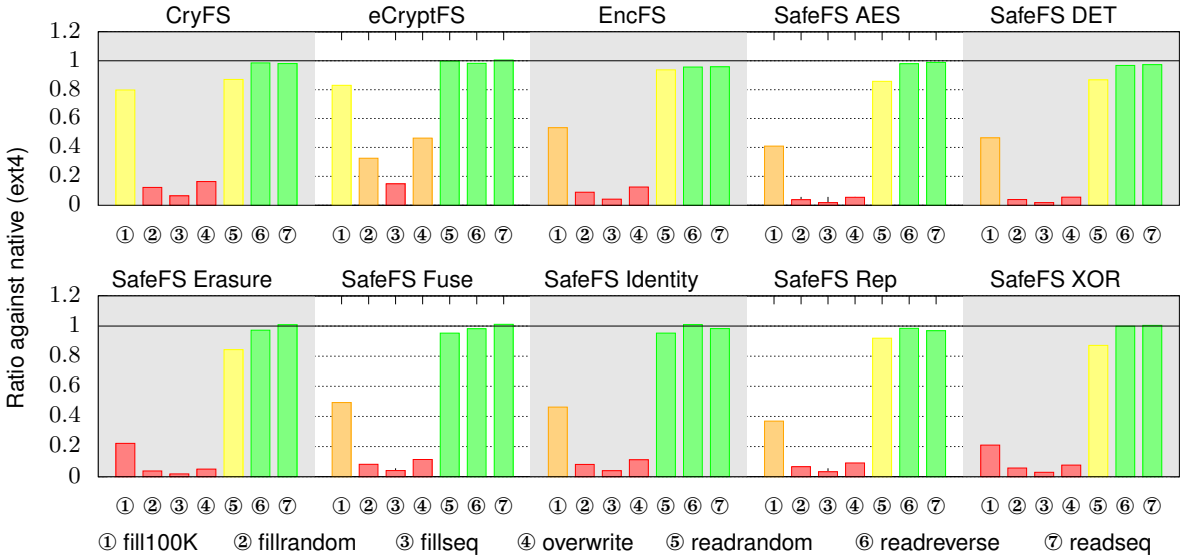


Figure 4.5: Relative performance of `db_bench` workloads against `native`.

backends we use NFS servers as it provide a standard file system interface that is ubiquitous on any cloud or external third-party service. The evaluation was done using OLTP-bench v1.0 (`oltpbenchmark`) [DPCCM13], a testbed for relational databases. This testbed contains more than 15 workloads that can stress a database system under different scenarios such as transactional, analytical, web-oriented and NoSQL applications. Additionally, the testbed collects not only the overall system throughput but also the resource usage of the remote servers as well as database statistics.

The experiments ran on a private cluster with 5 physical nodes, each with an Intel Core i3-7100 CPU, clock rate of 3.90 GHz and 2 physical cores with hyper-threading. All nodes had 16 GiB DDR3 RAM and a solid-state storage (Samsung PM981 NVME). The nodes were connected with a single 10 GiB network switch. We note that all of the software was deployed on bare metal without any virtualization or containers.

4.5.4 Micro-benchmark results

We ran several workloads for each considered file system (3 third-party file systems and 7 SafeFS stacks). The results have been grouped according to the workloads. First, we present the results of using `db_bench`, then `filebench` and, finally, we describe the results of running latency analysis for SafeFS layers.

db_bench evaluation. We first present the results obtained with `db_bench`. We pick 7 workloads, each executing 1M read and write operations on `LevelDB`, which stores its data on the selected file systems. The *fill100K* test (identified by ①) writes 100K entries in random order. Similarly, the entries are written in random order (*fillrandom*, ②) or sequentially (*fillseq*, ③). The

overwrite (④) test completes the write-oriented test suite by overwriting entries in random order. For read-oriented tests, we considered 3 cases: *readrandom* (⑤), to randomly read entries from the databases, *readreverse* (⑥) to read entries sequentially in reverse order, and finally *readseq* (⑦) to read entries in sequential order.

Figure 4.5 presents the relative results of each system against the same tests executed over a `native` file system (ext4 in our deployment). We use a colour scheme to indicate individual performance against native: red (below 25%), orange (up to 75%), yellow (up to 95%), and green ($>=$ 95%). We observe that all systems show worse performance for write-specific workloads (①–④) while performing in yellow class or better for read-oriented workloads (⑤, ⑥, and ⑦). The results are heavily affected by the number of entries in the database (*fill100K* ① vs *fillrandom* ②). As the size of the data to encrypt grows, the performance worsens. For instance, the SafeFS `XOR` configuration (the one with the worst performance) drops from 21% to 0.5%. The same observation applies for CryFS (the system with best performance) that drops from 79.78% to 12.33%.

The results for the *fillseq* ③ workload require a closer look as they have the worst performance in every file system evaluated. Since `db_bench` is evaluating the throughput of `LevelDB` which is storing its data on the evaluated file systems, it is necessary to understand an important property of `LevelDB`. The database is optimized for write operations, which results for *fillseq*, in high throughputs on native file systems contrasting with the selected file systems, where the throughput is significantly lower. As a matter of example, comparing throughputs for *fillseq* vs *fillrandom* on `native` (17.4 MB/s vs. 7.74 MB/s) and CryFS (1.14 MB/s vs. 0.94 MB/s) shows how much of the initial gains provided by `LevelDB` are lost.

While the processing of data heavily impacts the writing workloads, reading operations (⑤, ⑥ and ⑦) are relatively unaffected. The results for *readrandom* range from 87.05% with CryFS up to 99.81% for eCryptFS. Moreover, in experiments that switch the reading offset, the results are even better. In more details, the results never drop below 95.67% (*readreverse* on EncFS) independently of whether the reading is done from the beginning or the tail of the file.

Overall, the different SafeFS stacks perform similarly for the different database operations. The privacy stacks (see Table 4.1) performs comparably to the other file systems on most operations. Only the *fill100K* test shows significant differences, in particular against CryFS and eCryptFS. As expected, the deterministic driver provides a better performance (46.69%) against `AES` (40.96%) and *XOR* (20.98%). The redundancy stacks perform similarly. The erasure driver is slightly less efficient (22.11% of the native performance) due to the additional coding processing.

filebench evaluation. Next, we look at the relative performance of various workloads from `filebench`. Figure 4.6 depicts our results. We use the same color scheme as for `db_bench`. The seven workloads, executed over the different file systems and configurations, can be separated

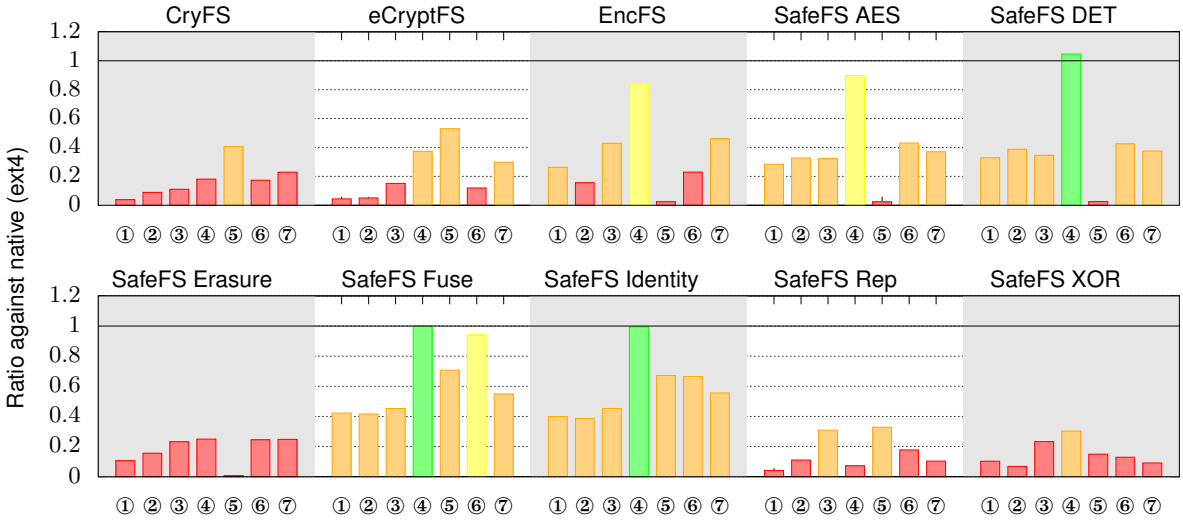


Figure 4.6: Relative performance of `filebench` workloads against `native`.

in two sets: application emulations (file-server ①, mail-server ②, web-server ③) and micro-workloads (④ to ⑦). This classification also introduces 3 major differences. First, the application emulation benchmarks last for 15 minutes, while the micro-workloads terminate once a defined amount of data is read or written. Second, the number of threads interacting with the system is respectively set to 50, 16, and 100 for the three workloads ①, ②, and ③, while micro-workloads are single-threaded. Third, the focus of micro-workloads is to study the behavior of a single type of operation while the application emulations usually run a mix of read and write operations.

In the micro-workloads (④–⑦), we observe the performance of the tested solutions in simple scenarios. Reading workloads (④ and ⑥) are most affected by the reading order. Surprisingly, our implementation performs better than the baseline with `DET` at 104.68% on random reads. These observations contrast with the results obtained for sequential reads where the best performing configuration in this case is SafeFS `fuse` (94.24%). On the writing side, micro-workloads ⑤ and ⑦ also display different results. For sequential writes (⑦), SafeFS `Identity` stack tops the results at 55.56% of the native performance. On the other end of the scale, SafeFS `XOR` stalls at 9.14%. The situation does however get a little better when writing randomly: `XOR` then jumps to 14.98%. An improvement that contrasts with the case of erasure coding (that has to read all the existing data back before encoding again) where the performance dramatically drops from 29.93% to 0.7% when switching from sequential to random writes.

On the application workloads side, the mixed nature of the operations gives better insights on the performance of the different systems and configurations. The systems that make use of classical cryptographic techniques consistently experience performance hurdles. As the number of write operations diminishes, from ① to ③, the performance impact decreases accordingly. Another important factor is the use of weaker yet faster schemes (such as re-using IVs for SafeFS `DET`). As expected, those provide better results in all cases. Indeed, `DET` tops at 38.74%

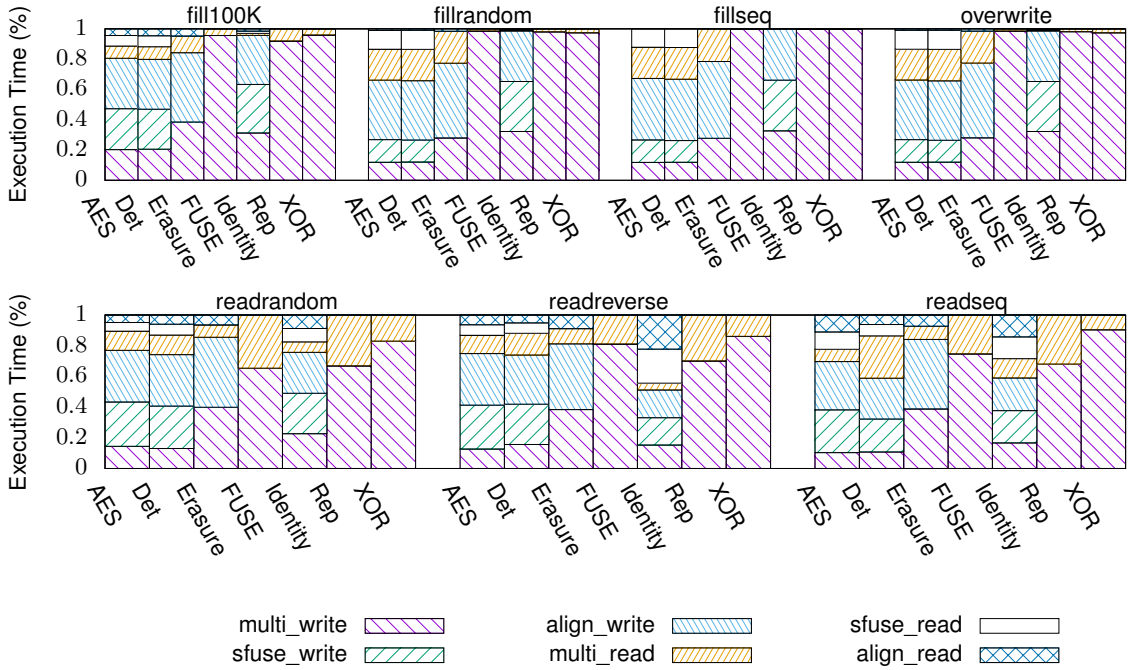


Figure 4.7: Execution time breakdown for different SafeFS stacks.

for ②. Resorting to more secure solutions can still offer good results with SafeFS `AES` (①: 28.40%, ②: 32.32%, and ③: 32.79%) but the need for integrated access control management should not be neglected for an actual deployment. The remaining SafeFS stacks also exhibit signs of performance degradation as the data processing intensifies.

Beyond the specifics of the data processing in each layer, the performance is also affected by the number of layers stacked in a configuration. As evidence, we observe that the `Identity` stack has a small but noticeable decrease of performance when compared with other `FUSE` stacks. Overall, the privacy-preserving stacks of SafeFS with a single backend have a better performance than the other available systems across the workloads. This benchmark suggests that user-space solutions, such as those easily implementable via SafeFS, perform competitively against kernel-based file systems.

Layers breakdown. In addition to using `db_bench` to study the performance degradation introduced by SafeFS, we use some of its small benchmarks (*fill100K*, *fillrandom*, *fillseq*, *overwrite*, *readrandom*, *readreverse*, and *readseq*) to measure the time spent in the different layers as the system deals with read and write operations. To do so, SafeFS records the latency of both operations in every layer loaded in the stack. The results obtained are presented in Figure 4.7. We note that for all these benchmarks, the initialization phase is part of the latency and that the time stacks show the sum of a layer's inherent overhead and the time spent in the underlying layers.

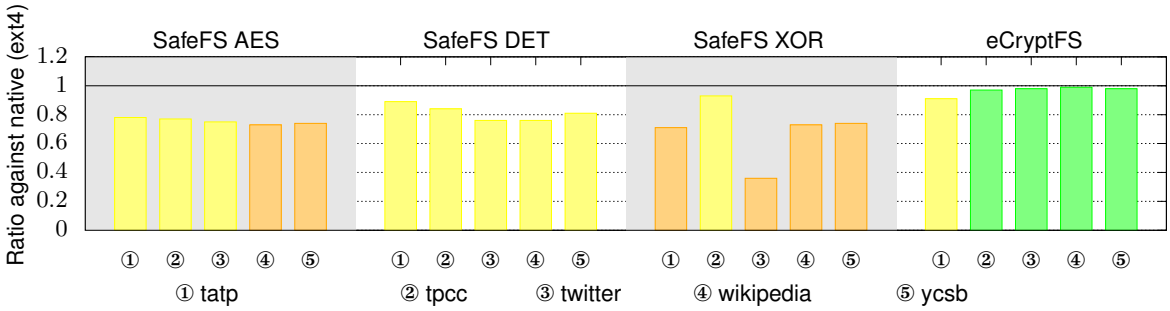


Figure 4.8: Relative performance of `oltpbench` workloads against `native`.

As expected, the time spent in each layer varies according to the tasks performed by the layers. The 3 most CPU-intensive stacks (`AES`, `DET` and `Erasure`) concentrate their load over different layers: `sfuse` for the first two and `multi` for the last one. Indeed, more time is spent in `multi`, the lowest layer, in non-privacy-preserving configurations. Another noticeable point is the increase in time spent reading data back for `Erasure` in the `multi` layer (11.03% for *fill100K*, 21.19% for *fillrandom*, and 21.53% for *fillseq*) compared to the decrease for `REP` (respectively 8.02%, 1.93%, and 0.05%) and `XOR` (4.03%, 2.59%, and 0.05%) stacks.

4.5.5 Macro-benchmark results

In the macro-benchmark we are concerned on measuring the overhead of outsourcing the data storage to a remote encrypted file system. As such, we focus our evaluation on the security stacks of SafeFS and compare the results to a native remote file system and to a kernel-space alternative, eCryptFS. The evaluation consists on measuring the throughput of 5 different workloads. Before evaluating a workload, the database is cleaned and initialized with a new scheme and fresh data records. From the set of workloads supported on `oltpbench` we selected two transactional workloads, two web-based workloads and a single NoSQL workload. For the transactional workloads and the web-based workloads we used a small and a large database size. All workloads were evaluated with a single client connection.

In transactional workloads a client submits sets of SQL queries to the database that either modify or manipulate the data and if a single query fails, the database has to be restored to a previous consistent state. The transactional workloads include the *tatp* (identified by ①) workload [Neu] that emulates a typical small home location register used by telecommunication providers and the *tpcc* (identified by ②) workload that simulates large warehouse-centric order application. The *tatp* transactions have no more than 3 queries and are mostly read-intensive whereas transactions in *tpcc* touch multiple tables and are write-intensive. The *tatp* workload was evaluated on a database populated with 14 GiB while *tpcc* was evaluated with 50 GiB.

The web-based workloads do not have transactions, instead the focus is on emulating social networks with database schemas that have many-to-may relationships and graph queries with non-uniform accesses. We selected two workloads that simulate popular social networks, the *twit-*

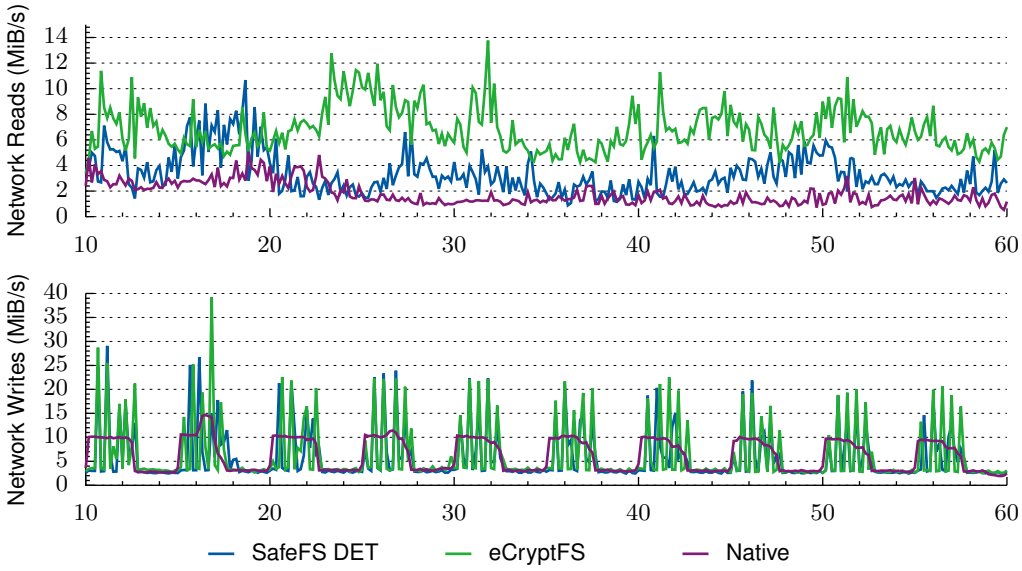


Figure 4.9: Network I/O of SafeFS `AES`, eCryptFS and native on `tatp` workload.

ter (identified by ③) workload [CHBG10] and the *wikipedia* (identified by ④) workload [UPvS09]. The foremost workload was evaluated with a database populated with 13 GiB while the latter had 47 GiB. We also evaluated an additional workload that captures NoSQL systems that have a simpler data model with just key-value operations. The *YCSB* (identified by ⑤) [CST+10] workload simulates applications that require high-scalable NoSQL databases. The database was populated with 16 GiB.

Figure 4.8 presents the results of the macro-benchmark evaluation with 1 hour runs for every combination of system and workload. The results presented is the ratio of the throughput in comparison to the `native` file system and we use the same color scheme as the previous benchmarks. Across every workload, the SafeFS stacks have similar performance with the exception of the SafeFS `XOR` stack. The encoding and distribution of data leads to a greater performance difference. This difference is mostly prominent in the smallest datasets with `twitter` having the highest performance decrease with an overhead 64%. In contrast, the SafeFS `XOR` stack has the smallest overhead in the workload `tpcc` from all of the SafeFS stacks with just a 7% performance slowdown. In alignment with the previous results, the SafeFS `DET` stack has a better performance than SafeFS `AES` and has a relative performance ratio between 76% and 89% across every workload. However, there is no clear performance advantage of using SafeFS `DET` over SafeFS `AES` in the web-based workloads as there is at most a negligible difference of 3% in the `wikipedia` workload. The `ycsb` workload has almost a constant overhead in every SafeFS stack that varies between 74% and 81%.

These results show that kernel-space solutions, in particular eCryptFS, have a small but considerable advantage over SafeFS in this macro-benchmark setting. As observed, in almost every workload eCryptFS has at most a 3% performance overhead in contrast to the native.

However, in the `tatp` workload this overhead increases to 9%. In fact, when compared to SafeFS `DET`, eCryptFS is only 2% faster. Figure 4.9 provides a in-depth look at the network usage of both systems as well as to the native network file system. We observe, that the network write traffic of SafeFS `DET` overlaps with eCryptFS and both follow the same pattern as the native file system. However, the write traffic of SafeFS `DET` stabilizes between 2 to 8 MiB/s after 30 minutes whereas eCryptFS is consistently writing more than 4 MiB/s. Depending on the information stored by applications similar to `tatp` it might be an acceptable security trade-off to use SafeFS `DET` over eCryptFS due to the lower resource usage and the similar performance.

4.6 Summary

In this Chapter we addressed the problem of outsourcing a database storage to an untrusted third-party while processing queries on a trusted client. We address this problem with a high-level approach that abstracts the application and protects data at the file system level. Existing systems are divided in kernel-based solutions and user-space solutions, but state-of-the-art solutions are monolithic and only support a limited set of encryption schemes. We propose SafeFS, a modular user-space Fuse-based architecture that allows applications to compose functional components (*layers*), each with a specific feature (*drivers*). This modular and flexible design allows extending layers with novel algorithms in a straightforward fashion, as well as reusing existing Fuse-based implementations. SafeFS layers can be used to compress, encrypt and replicate applications data without any intrusive modification to applications.

We compared several SafeFS stacking configurations against state-of-the-art systems and demonstrated the trade-offs for each of them. Our extensive evaluation based on real-world benchmarks, including multiple database workloads, show that our user-space approach has a practical performance and is in fact a suitable alternative to more efficient kernel-space solution. Database benchmarks show that SafeFS most efficient privacy stacks have at most 26% overhead in comparison to a native NFS remote file system. We see two future lines of work in SafeFS. We envision a context and workload-aware approach to choose the best stack according to each application's requirements (*e.g.* storage efficiency, resource consumption, reliability, security) leveraging SDS control plane techniques that enforce performance, security, and other policies across the storage vertical plane stack [TBO+13]. For instance, applying stacks on a per-file basis, meaning that SafeFS encrypts only sensitive tables of a database system instead of using the same privacy stack to every file and folder. Security-wise, SafeFS still discloses some information to an untrusted server such as the which files are most accessed, which blocks are read/updated and how many times they are touched. While there are encryption schemes that address this problem, their integration as new layers is not straightforward as they require multiple communication rounds with the untrusted server. Furthermore, a naive integration can result in a prohibitive overhead and there may be significant optimizations left

unexplored that are application specific.

Chapter 5

A trusted NoSQL database on untrusted clouds

In this chapter, we present our second main contribution, a multi-cloud NoSQL framework that protects the users confidentiality by distributing data as well as query processing on multiple non-colluding parties. This works adds an additional security layer on top of the encrypted storage presented in the previous chapter and advances even further the idea of leveraging multiple cloud providers. The results obtained with this work present a novel trade-off between privacy and performance in the state of the art of CPD by leveraging secure multiparty protocols.

5.1 Introduction

Offloading only the database storage to an untrusted site using encryption is a prudent choice for the most privacy-sensitive applications. By processing queries on a trusted private site and never disclosing the encryption key, the client remains in control of the database. However, to benefit from the cloud paradigm to the full extent applications have to outsource computation securely. Recent efforts have tackled this challenge with PPE schemes such as CryptDB [PRZB11] and SQL Server Always Encrypted [Mica] which are considered the state-of-the-art on CPD.

However, these systems have a few issues, as the same security guarantees that make PPE schemes attractive end up compromising the user's confidentiality. In fact, if a malicious external attacker gains access to a data dump of a database they can learn sensitive information even if the data is encrypted. For example, a common attack vector consists in performing frequency analysis on datasets without a uniform distribution. Notably, prior work has shown that, depending on the dataset, more than 50% of the data can be disclosed by correlating information from different columns [DDC16]. Inference attacks are also effective at disclosing sensitive data with more than 90% accuracy from a single database snapshot encrypted with

PPE schemes [BGC+18].

In this chapter we explore a different approach to centralized privacy-aware database that use multiple cryptographic schemes. We propose d'Artagnan, a secure multi-cloud NoSQL database framework that decentralizes information by encrypting data in secrets and storing them in independent databases, each hosted on a different cloud provider. A single secret does not leak any information and the original plaintext value can only be decrypted if the majority of the secrets are obtained by a single entity (e.g.: client application). Furthermore, with these secrets d'Artagnan can process any query on the server-side by evaluating secure cryptographic protocols. An immediate result of such system is that a data breach of a single cloud is inconsequential and, for any significant information to be obtained, the majority of the clouds that constitute d'Artagnan must be compromised. Clearly, the effort required to break the system grows with the number of cloud providers used.

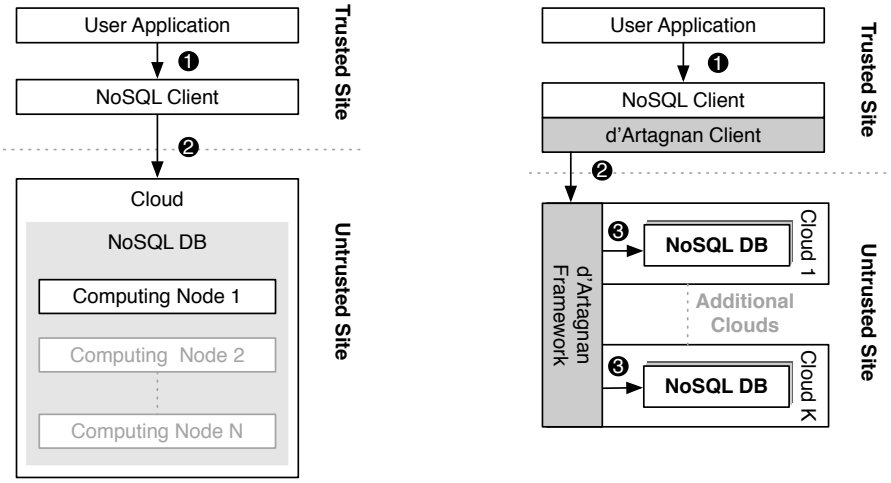
At the core of d'Artagnan are secret sharing schemes and secure multiparty protocols. Secret sharing schemes are used to encrypt data in multiple secrets while SMPC protocols enable the database to process queries over the secrets. These cryptographic techniques have been overlooked by SQL and NoSQL databases as a practical alternative to PPE schemes despite being capable of evaluating any function and supporting passive as well as active adversaries. Furthermore, protocols remain secure until a predefined threshold of parties are corrupted (e.g.: the majority of the parties). However, leveraging these protocols on a database requires addressing some common distributed system challenges such as establishing a network of nodes, discovery nodes and supporting dynamic removal or addition of new nodes.

We address this gap with the design of a high-level, modular and extensible multi-cloud database framework that encrypts data in secrets and process queries with SMPC protocols. The system hides the details of the underlying cryptographic protocols and the distributed execution of secure queries under a standardized NoSQL API. Note that we chose to focus on key-value NoSQL systems as they provide a small kernel of operations that can be composed to create richer queries similar to SQL databases.

This chapter has 6 sections, starting with Section 5.2 that defines a general model of the problem addressed. Section 5.3 presents d'Artagnan's architecture and Section 5.4 describes how every component in the system interacts to securely process queries. Section 5.5 describes the prototype implementation. Section 5.6 presents and discusses the experimental evaluation. Finally, Section 5.7 provides a discussion of this chapter.

5.2 Problem definition

In this chapter we address the problem of securely outsourcing computation as well as storage of a NoSQL database to multiple third-party services. More precisely we limit the problem to



(a) Plaintext NoSQL Database (b) d'Artagnan Approach

Figure 5.1: System models of a plaintext NOSQL database and the d'Artagnan approach.

key-value NoSQL database systems. These databases are suitable for applications that handle eventually consistent data, support lack of database-wide transactions and require a flexible data scheme [Cat11]. Conceptually, data is stored in named multi-dimensional maps, similar to hash tables, where values are indexed by a unique key and a column. The map data structure does not follow a static schema and is dynamically adjusted as new rows are inserted. Clients interact with the database to store or process data through an interface similar to the following [Mel98]:

Put$(Table, Column, Key, Value)$**:** Given a table, store a value indexed by the row key and column.

Update$(Table, Column, Key, Value)$**:** Given a table, update the record indexed by the row key and column with the specified value.

Get$(Table, Column, Key)$**:** Retrieve the value of a table indexed by a row key and a column.

Delete$(Table, Column, Key)$**:** Delete the value of the table indexed by a row key and a column.

Scan$(Table, Start, Stop)$**:** Retrieve the records of a table whose index is greater than or equal to the identifier $Start$ and lower than the identifier $Stop$.

Filter$(Table, Condition)$**:** Given a table, retrieve all the records that validate the condition predicate. The condition can specify columns, propositional formulas, and regular expressions.

In Figure 5.1a we depict a high-level model of the problem of outsourcing a NoSQL database to an untrusted service. This model is divided in a *Trusted Site* and a *Untrusted Site*. On the first part resides a user application (e.g.: web-server) that interacts with the NoSQL database with a native NoSQL client (Figure 5.1a-❶). The NoSQL client exposes an interface similar to the one

previously defined. Application requests are intercepted by the NoSQL client and forwarded to the NoSQL database (Figure 5.1a-❷) deployed on a *single* cloud.

On the *Untrusted Site*, the NoSQL database architecture consists of a distributed system with multiple nodes. Generally, NoSQL databases trade strict consistency and transactional support for a highly scalable, distributed architecture [Cat11, CDG+08]. This trade-off is key to design "shared nothing" databases that scale horizontally by partitioning data into *shards*, range of table rows uniquely defined by an initial row key and a final row key. Shards are created as a table grows, replicated and balanced between a pool of *computing nodes*. Every shard is assigned to a single node, but a node can manage multiple database shards. The mapping between a shard and a node is considered meta-data managed by a proxy master node. However, this proxy node is only used sporadically by database clients to find nodes that contain specific regions. Queries are directly sent to *computing nodes* without any middleman. In case of a node failure, its shards are reassigned to the remaining live nodes which handle incoming requires while a new node is added to the cluster. Nodes are stateless and can be considered individual processing units that do not share any in-memory context between each other.

We now give an intuition on our approach to cryptographically protect data. We depict our approach in Figure 5.1b but provide a more detailed and extensive description of our architecture in the following sections. Whereas current systems store encrypted data on a single system, we split data by multiple K NoSQL databases. We keep the division between *Trusted Site* and a *Untrusted Site* but we add to each site a d'Artagnan component. On the *Trusted Site*, the user application sends and receives requests as plaintext (Figure 5.1b ❶). When a request is intercepted by the d'Artagnan client than data is encoded into multiple shares (Figure 5.1b-❷). The shares are forwarded to the d'Artagnan framework that is deployed on the *Untrusted Site* alongside K clouds. The d'Artagnan component stores the shares on the NoSQL databases (Figure 5.1b-❸) and retrieves them to process queries as necessary. The number of clouds is fixed, and each cloud contains an independent NoSQL database that has a varying number of N computing nodes. The number of nodes inside each database can be dynamically adjusted depending on the application workload. The d'Artagnan framework ensures data stays consistent across the clouds and that queries are correctly evaluated. To ensure no single component besides the NoSQL client has all the share, the d'Artagnan framework is in fact a distributed component split across the clouds.

5.2.1 Trust model

In a N-cloud system, d'Artagnan encrypts sensitive values with a (N, t)-secret sharing scheme and stores each share on a single database. The division of data in shares ensure that *data at rest* remains secure as long as no more than t clouds are compromised. Even if a subset of the cloud providers leak multiple database snapshots, it is proven to be impossible to decrypt the

original values [Sha79].

Besides protecting data at rest, d'Artagnan's goal is to protect user's confidentiality during query processing. In this case, d'Artagnan security guarantees depend on the properties and trust model of SMPC protocols. d'Artagnan is designed to leverage different SMPC protocols to protect user's data from external attackers and malicious insiders. We assume the simplest adversary supported by the majority of SMPC protocols. We consider a static, semi-honest adversary that can corrupt a threshold of the clouds. For each corrupted cloud, the adversary has access to all of the database *computing nodes*, the messages received and sent by the nodes as well as the request access patterns, the storage access patterns, the data stored in memory and persisted to storage. However, the adversary does not have access to anything that happens on the *Trusted Site* and cannot corrupt more nodes than the predefined threshold at any moment. Stronger adversaries can also be supported by d'Artagnan such as active or dynamic adversaries if the appropriate multiparty protocol is used.

5.3 Architecture

d'Artagnan's architecture, depicted in Figure 5.2, coordinates independent key-value databases. Each database is hosted at different cloud providers and d'Artagnan's architecture creates a logical NoSQL database capable of processing queries over encrypted data. This decentralized approach prevents a single cloud provider or a database vulnerability from corrupting and compromising the entire system's security guarantees. When taken to the limit, it is entirely possible to have a system deployment where the first cloud (Party 1) is a BigTable database on Google's Cloud, the second cloud (Party 2) hosts a Dynamo DB in Amazon AWS and the remaining clouds host entirely different databases.

From a high-level perspective, the framework operates across a *Trusted Site* and an *Untrusted Site*. The *Trusted Site* is the system's entry-point where the client application resides sheltered from attacks. This site, possibly a private trusted infrastructure, is where sensitive data is encrypted before being outsourced to the cloud. The *Untrusted Site* is where all the data storage and query processing take place. This site has two or more clouds, each playing the role of a SMPC protocol party. Every party (cloud) hosts an autonomous key-value NoSQL database which has no knowledge of the remaining parties. Without d'Artagnan's components, the party's databases are nothing more than a storage system that holds encrypted data. This section describes these components and their role on the framework architecture.

5.3.1 Trusted site

The first component is the **Safe Client**, a privacy-preserving layer between the *Trusted Site* and *Untrusted Site*. This component has two main goals, abstract the multiple underlying

NoSQL databases and protect sensitive data. The first goal is accomplished by exposing an high-level NoSQL API, as defined in Section 5.2, that enables applications to use d'Artagnan as a single NoSQL database. The high-level queries are transformed into multiple low-level requests sent to the *Untrusted Site*. However, before any query is sent to the *Untrusted Site*, `Safe Client` encrypts sensitive user information. The client can specify, on a column basis, the SMPC protocols used to process queries. For instance, a client can choose to protect a subset of table columns with protocols that ensure *confidentiality* against an active adversary while another subset is protected with protocols that just provide *confidentiality* against a semi-honest adversary. To ensure different security guarantees each protocol may implement a different secret sharing scheme. As such, `Safe Client` delegates the encryption and decryption of data to external software libraries that implement the following API:

Encode$(secret, n, k) \rightarrow [shares]$**:** Encode a secret in n shares such that only k shares are required to decode the secret. This function returns a list of shares.

Decode$([shares]) \rightarrow secret$**:** Given a list of shares, decode them and return the secret.

5.3.2 Untrusted site

Safe Server. The `Safe Server` component is the processing engine of the *Untrusted Site*. This component is a distributed layer with multiple nodes, at least one node per party, that intercepts high-level `Safe Client` requests and converts them into secure operations. The conversion process depends on the client request, the security model and the underlying databases, but it ensures that every client key-value NoSQL query is converted into a sequence of database-specific requests and SMPC protocols that have a semantically identical functionality with additional security guarantees. Overall, the `Safe Server` initializes the necessary resources and leverages every available component, the `Multiparty Library`, `Network Middleware`, `Discovery Service` and the underlying database to securely process client requests. A detailed description of the query transformation process and interaction of each component is presented in Section 5.4.

Multiparty Library. The `Multiparty Library` component contains the implementation of the secure multiparty protocols. This component abstracts the details of protocol implementations from the `Safe Server` with a high-level interface. Let $p = \{s_{p0}, s_{p1}, \ldots s_{pn}\}$ and $k = \{s_{k0}, s_{k1}, \ldots s_{kn}\}$ be two secret shared values and $p \circ k$ a comparison operation such that $\circ \in \{=, <, >, \leq, \geq\}$. The multi-party library API supports the following operation set $\bullet$:

- **Equal**(s_{pi}, s_{ki})

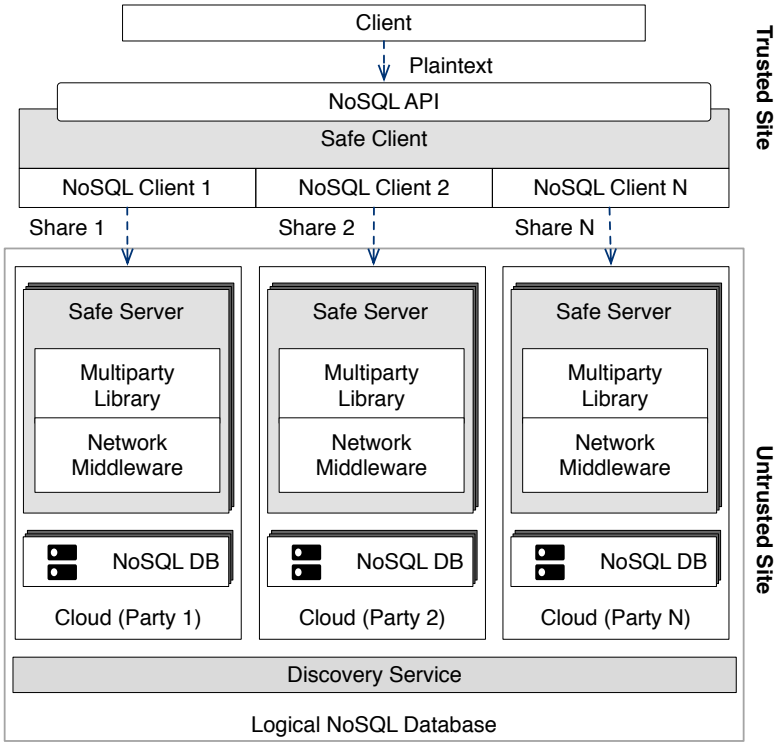


Figure 5.2: d'Artagnan architecture has two parts: a *Safe Client* and a *Logical Database*. The Safe Client resides on the *Trusted Site* and intercepts client requests. The Logical Database consists of independent NoSQL databases hosted at independent cloud providers. Cloud providers are considered an *Untrusted Site*. In both parts, the gray boxes refer to d'Artagnan components and white boxes represent unmodified third-party components.

- **LessThan**(s_{pi}, s_{ki})
- **GreaterThan**(s_{pi}, s_{ki})
- **LessThanOrEqualTo**(s_{pi}, s_{ki})
- **GreaterThanOrEqualTo**(s_{pi}, s_{ki})

such that $p \circ k \equiv s_{pi} \bullet s_{ki}$. This interface has a set of core operators capable of satisfying the high-level key-value NoSQL API. However, this set is extensible and can support additional application-specific functions.

This library also defines a clear boundary between the database execution and the SMPC protocols' implementations. It encapsulates the details of the NoSQL database from the protocol implementations, enabling expert cryptographers to integrate new protocols without having to write database-specific logic. All of the necessary context to integrate new protocols is provided by an execution environment that contains a `Safe Server` party ID and a `Network Middleware` client. The party ID determines how the library evaluates a protocol circuit and the `Network Middleware` client enables the parties to exchange shares.

Network Middleware The `Network Middleware` component establish a mesh network between every party and ensures the protocols exchange shares correctly. The parties can evaluate protocols and communicate via the following network interface:

Send$(playerID, share)$**:** send a share to a player.

Receive$(playerID)$**:** receives a share from a player.

This simple interface can support a wide range of SMPC protocol implementations and abstracts the `Multiparty Library` from concurrent protocol executions. At any time, a single `Safe Server` node can process multiple concurrent requests and start parallel SMPC protocols in the `Multiparty Library`. If concurrent protocol executions have to send shares from party A to party B, the `Network Middleware` multiplexes the shares sent by party A and forwards them to the correct protocol execution environment in party B. With this approach, the protocols in the `Multiparty Library` only need to implement the circuit's logic without having to deal with concurrency issues.

Besides concurrent protocol execution, the `Network Middleware` also handles the execution of protocols in a dynamic setting. The set of *computing nodes* in a NoSQL database that play the role of parties in a protocol is not static and can be added or removed as necessary. For instance, the set of available nodes can increase to improve the system availability and performance while ensuring the same security guarantees. As such the `Network Middleware` adapts to the dynamic set of nodes and routes the shares to the correct participants by leveraging the `Discovery Service`.

Discovery Service The `Discovery Service` component keeps track of the `Safe Server's` nodes status and location. Every time a node goes online in any of the clouds it notifies the `Discovery Service` with a payload. The payload contains meta-data regarding the node, such as the node's IP address and the database tables it manages. When a node goes offline, it reports its new status to the service. None of the information regarding the shared secrets is kept on the `Discovery Service` and the connections between the nodes is made directly through the `Network Middleware` without using the `Discovery Service` as a proxy. d'Artagnan makes no assumptions on the actual implementation of the `Discovery Service`. Since the meta-data contains no sensitive information, it can be stored on a distributed NoSQL database to avoid a single point of failure and deployed in any public or private infrastructure.

5.4 Secure query processing

This section describes how d'Artagnan processes queries securely and how every component interacts. We use a **Filter** operation as an example. Figure 5.3 depicts an illustrative deployment scenario with N clouds used throughout this section. The first cloud hosts a BigTable database whereas cloud N cloud hosts an HBase database. In this example, d'Artagnan stores a Table T with M columns, from column C_1 to column C_M. In this table only the columns store sensitive data and the row keys are simple plaintext identifiers (e.g: $K1$ and $K2$ in Figure 5.3).

Consider a client plaintext Filter$(T, C_1 == D)$ request on table T to search for every record where the column C_1 has the value D. The table can have several records matching this condition, but, for simplicity purposes, let us consider that only the record with key $k4$ (last row of table T) has value D. Every value stored in the tables is encrypted in shares, with each database storing a single secret. The `Safe Servers` task is to find its party's secrets of the matching rows and send it to the `Safe Client`. Upon receiving all secrets, the `Safe Client` can disclose the result and send it to the client.

The first step in d'Artagnan's execution flow consists in protecting the client's plaintext Filter (Figure 5.3-❶). Upon receiving the request, the `Safe Client` has three main tasks. First, it checks the client's security requirements to select the secret sharing scheme to use. Secondly, it encrypts the search value D in N shares with the proper secret sharing scheme. Finally, it generates N secure Filters, each containing a single secret instead of the client's plaintext values. Afterwards, the filters are sent in parallel (Figure 5.3-❷) to the cloud providers, one filter per party. It's worth noting that the encrypted shares are always random and not deterministic. As such, the shares generated in the `Safe Client` for the value D are different from the shares stored in the databases for the same value. The only way the database can process the incoming request is with an SMPC protocol.

Every party's `Safe Server` intercepts the secure queries and start an execution flow to process the secure Filter. The flow is identical in every party and executed in parallel by each

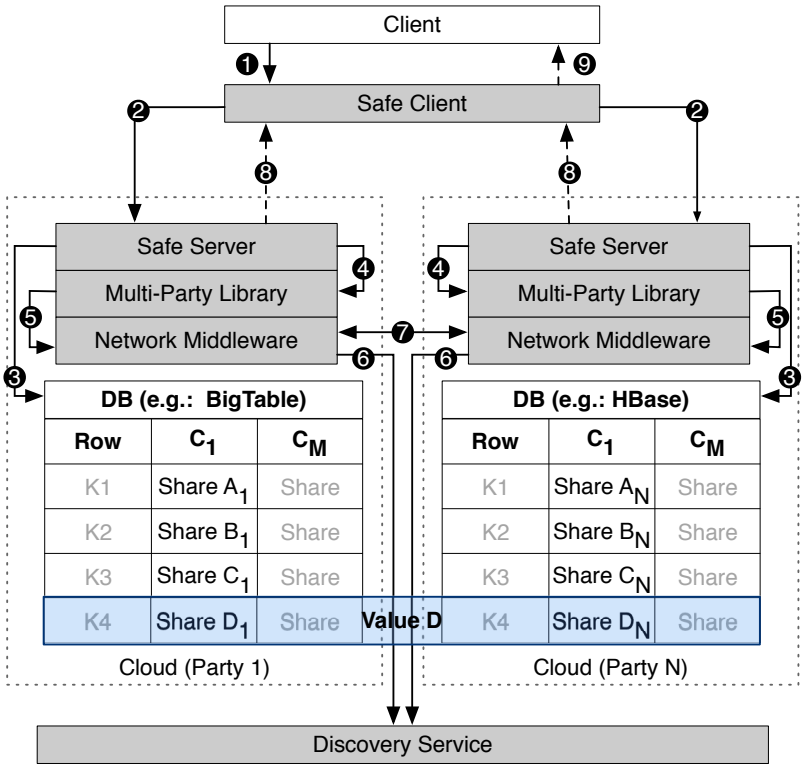


Figure 5.3: Interaction of d'Artagnan components to process a client request.

party's `Safe Server`. Thus, we describe the execution flow from the viewpoint of party 1 depicted in Figure 5.3. Upon intercepting the request, the `Safe Server` scans the database (BigTable) in batches. In this example there is only a single batch, from $K1$ to $K4$ (Figure 5.3-❸). Afterwards, it creates an execution environment containing its player ID (1) and a `Discovery Service` client. With the execution environment, the `Safe Server` is ready to find the records that satisfy the Filter condition and starts an Equal protocol (Figure 5.3-❹) with the shares stored in column C_1.

The `Multiparty Library` starts to participate in the protocol execution by evaluating the Equal encrypted circuit (Figure 5.3-❺). Eventually, every party's `Safe Server` reaches this point and also starts to participate in the protocol by invoking the same function with their own shares. The compiled Equal circuit is composed of multiplication gates that requires parties to exchange shares during the circuit evaluation. However, before any share can be sent, the `Network Middleware` contacts the `Discovery Service` to store a payload signaling its location and the request it's processing (Figure 5.3-❻). This information enables each party's `Safe Server` to find its peers in the computation. For instance, if party 1 has to send a share to party N, the `Network Middleware` asks for the IP address of the `Safe Server` in party N that is processing an Equal protocol on table T. In a real execution, additional information is required since there can be multiple concurrent protocols being executed. However, for the `Multiparty Library` these details are abstracted by the `Network Middleware`. When the parties learn each other's locations, the party's `Network Middleware` establish a communication channel and exchange the necessary shares to evaluate the protocol (Figure 5.3-❼).

The protocol evaluation either completes successfully or aborts. Protocols abort if the implementation ensures *fairness* guarantees and an active attack is detected. Moreover, a protocol with *identifiable aborts* [CL17] returns the party ID of the corrupted party. In this scenario, the information is sent to the client that decides the proper course of action. A successful protocol evaluation returns the rows satisfying the Filter request. In the example, it's the row with key $K4$ depicted in Figure 5.3. An attacker also learns this information if a party is corrupted. Even though this information by itself is not sufficient to break the system's confidentiality, it's an open problem that can be addressed with protocols that ensure oblivious execution [KS14a]. The server-side processing ends with every `Safe Server` sending the shares of the rows that satisfy the protocol to the `Safe Client` (Figure 5.3-❽) which discloses the original row values and returns the correct result to the client (Figure 5.3-❾).

5.5 Implementation

We implemented a complete and fully-functional prototype. This prototype supports Apache HBase, a scalable, open-source NoSQL relational database [hba]. HBase stores data in a multi-dimensional map similar to the NoSQL data model previously presented in Section 5.2. However, tables are partitioned automatically by the system in multiple shards. A shard is a

storage unit of a subset of consecutive tables rows that enables the system to scale horizontally by balancing the workload between the database's computing nodes.

A single HBase deployment is a distributed system with two types of nodes: a single Master node and multiple RegionServers (computing nodes). The Master stores meta information regarding cluster configuration and database tables. The RegionServers stores and processes all of the database data. Each RegionServer hosts a set of shards and each shard can only be served by a single RegionServer. The query execution is handled directly between clients and RegionServers.

d'Artagnan's components are implemented in Java. The `Safe Client` prototype provides the same API as the HBase client but manages multiple HBase client connections, one for each party. As described in Section 5.3, this component transforms plaintext HBase queries into secure HBase requests. However, the `Safe Client` prototype contains a pool of threads that executes the secure requests in parallel. The results of the secure requests are decrypted and aggregated in a single final result forwarded to the client application.

The `Safe Server` component is integrated as an HBase coprocessor, a feature that enables developers to extend RegionServers behavior with plugins. The coprocessors intercept every request sent to a RegionServer and have access to an internal database API capable of modifying the RegionServers core behavior. This approach brings secure query processing closer to the data by removing a network hop between the `Safe Server` nodes and the parties databases.

In this prototype a `Safe Server` node is instantiated per RegionServer. Since each HBase database cluster can have more than a single RegionServer, the role of a single SMPC party is in fact played by multiple `Safe Server` nodes. To manage the multiple nodes as a single party, the `Safe Server` nodes store a payload on the `Discovery Service`, implemented as a Redis [Red] database. The payload contains the address of each `Safe Server` node, the party it belongs to and a unique request identifier for each client query. This information enables the `Network Middleware` to discover which `Safe Server` nodes are evaluating an SMPC protocol and establish a mesh network. Even when one or more parties have multiple `Safe Server` nodes, each processing concurrent requests, the `Network Middleware` is able to route shares between the nodes participating in a protocol. The information stored on the `Discovery Service` reveals no information on the actual shares as the communication between the parties is made directly through TCP channels without using the `Discovery Service` as a proxy. Furthermore, all of the data stored on the `Discovery Service` can be cryptographically signed to prevent a malicious attacker from corrupting the information.

The `Multiparty Library` is implemented in Java and currently supports the protocols proposed by Bogdanov *et al.* [BNTW12]. These protocols are among the most efficient in the state-of-the-art and are one of the few protocols applied in the industry to protect critical information [BLW08]. This protocol suit is optimized for three independent parties with a single dealer and ensures confidentiality against a semi-honest adversary.

d'Artagnan inherits the security guarantees of the underlying secure multiparty protocols. As such, this prototype protects the user's confidentiality against malicious insiders and external attackers that do not modify the protocol execution. Although the current prototype does not support protocols that protect the user's confidentiality against an active adversary, new protocols can be integrated to protected data from such adversary. For instance, the current framework can easily leverage protocols such as the SPDZ Protocol Suite [SB18] which detects the presence of a corrupted party. Supporting these protocols only requires writing two wrappers: the wrapper for the encode and decode functions in the `Safe Client`; the protocols wrappers for the `Multiparty Library`. The remaining components, the `Safe Server`, `Network Middleware` and `Discovery Service` are orthogonal to the underlying protocol implementations and require no modifications.

5.6 Evaluation

This section presents the evaluation of d'Artagnan's prototype in two experimental settings, including a complete system deployment on public cloud providers. Furthermore, it demonstrates the current performance bottleneck of one of the most efficient secure multiparty protocols protocols with an industry-proven benchmark.

5.6.1 Experimental setup

Methodology. The evaluation measures d'Artagnan throughput (OP/s), and latency in two different settings. The first is a controlled setting of a fully distributed deployment designed to establish the system's performance baseline. The second scenario validates the prototype with a real-world deployment on the leading cloud providers: Google Cloud Platform [Goo], Microsoft Azure [Micb], Digital Ocean [Dig] and Amazon AWS [Ama]. HBase is the baseline used in both scenarios.

Use case. We use a synthetic use case of a medical clinic with an appointments table. The table contains the columns: *Physician ID*, *Patient ID*, *Date*, *Type* and *Institution ID*. Records in this table contain private data that must be protected. The information stored on the table reveals the location (*Institution ID*), *Type* (e.g.: Cardiology, Oncology) and date of an appointment. Furthermore, it leaks the relation between a physician and a patient. To protect this information, columns are encrypted with an additive (3,1)-secret sharing scheme. The table records are indexed by a numeric identifier (Key) stored as plaintext. These identifiers reveal no information and are only used to access the data.

Workloads	Get	Update	Insert	Scan	R-M-W	Filter
A	50%	50%	-	-	-	-
B	95%	5%	-	-	-	-
D	95%	-	5%	-	-	-
E	-	-	5%	95%	-	-
F	50%	-	-	-	50%	-
G	40%	20%	20%	-	-	20%

Table 5.1: Benchmark workloads. Each workload issues a request with the NoSQL API presented in Section 5.2. The operator $R - M - W$ is the composition of a Get and Update operation.

Benchmark. For a standard evaluation we use the industry-proven Yahoo! Cloud Serving Benchmark (YCSB) [CST+10]. This benchmark has six standard workloads (A-F), each with a different read-write ratio and value distribution. Following the approach commonly seen in related literature [KTV+18, CST+10], we omit the results of workload C as they are identical to the results obtained in workload B. The operators on these workloads only process the plaintext keys and do not measure the overhead of SMPC. However, the workloads are essential to evaluate d'Artagnan's overhead of protecting data with a secret sharing scheme and issuing concurrent requests to multiple clouds. Furthermore, it establishes a direct comparison with a standard HBase deployment without any secure processing. We designed a new workload G to measure the overhead of SMPC protocols. The workload simulates a clinic use case where the staff frequently browses through the daily appoints but sporadically schedules or updates a new appointment. The workload filters through the $Type$ of appoints using SMPC protocols. The request distribution of every workload is presented in Table 5.1.

One important aspect to the entire evaluation is the value distribution on the Appointments table and how the values for the NoSQL operations are chosen by the YCSB benchmark. On both scenarios the table identifiers are integer values that increase monotonically with every new row. Each table column value is sampled from a uniform distribution of the data type. The same applies for the input values of the Put, Update and Get operators. The Scan and Filter operators choose a starting key from a uniform distribution and iterate over every row until the last table row. The operator read-modify-write (R-M-W) presented in Table 5.1 is the composition of a Get and Update operation.

5.6.2 Controlled setting

Experimental set-up. d'Artagnan's prototype was deployed on a private infrastructure divided in three independent clusters. Each cluster is in itself a distributed HBase deployment with 2 Region Servers, each one with a `Safe Server`. In total, this deployment consisted of 10 nodes, 3 per HBase cluster plus the client machine with the YCSB benchmark that contains the `Safe`

`Client`. Every node had an i3 CPU 4 cores at 3.7 GHz, 8 GB of main memory and a 128 GB SSD. Hosts were connected over a shared Gigabit Ethernet network with an average latency of 0.3 ms.

YCSB workloads. Figure 5.4 presents the results of the YCSB workloads with the Appointments table containing 1 million rows divided between 20 shards. Each plot has latency versus throughput curves that depict the systems scalability with an increasing number of clients, from 1 to 256 in a logarithmic scale base 2. Each dot ($\times$, $\circ$) in a plot represents an experiment with a different number of clients. Experiments consists of 3 hour runs.

From workload A to F d'Artagnan prototype scales with the number of clients but starts to reach a plateau with 32 clients as requests latency increase at a higher rate than the throughput. For instance, on Workload A d'Artagnan prototype has a throughput increase of 10% from 32 clients to 256 clients but the latency increases 87%. Both d'Artagnan prototype and HBase follow a similar pattern across the workloads with a higher throughput on read-intensive workloads. The highest throughput reached by both systems is found on workload B with d'Artagnan prototype peaking around 17 KOP/s and HBase at 49 KOP/s. With a maximum overhead of 2.88$\times$ the baseline, d'Artagnan's overhead is acceptable considering that for each client, the `Safe Client` has to encrypt every column value with a secret sharing scheme, send three concurrent requests, one per cloud, and wait for all parties to process the request. On the maximum load of 256 concurrent clients, d'Artagnan has 768$\times$ more requests to manage than the baseline.

Workload G on Figure 5.4 is the first workload to measure the impact of SMPC protocols. Even though only 20% of the workload operations are filters that require SMPC, the protocols have a significant impact on the system throughput. d'Artagnan prototype is limited to 5 OP/s and reaches an average latency of 78 seconds with 32 clients. HBase scales with the increasing number of clients and peaks at 33 OP/s with an average latency of 10 seconds with 32 clients. With the highest throughput of both systems, d'Artagnan prototype is 6.6$\times$ slower than HBase. Even with the specialized SMPC protocols, d'Artagnan prototype main bottleneck is the network bandwidth used to evaluate the multiplication gates of the encrypted circuits, as depicted with the cumulative distribution function (CDF) in Figure 5.5. The presented CDFs' results were collected during the execution of Workload G with a single client on both systems. A single database client is sufficient to saturate the network bandwidth with just 10% of the d'Artagnan network usage bellow 140 MB/s. Computational resources are not a critical factor as the CPU usage in every experiment, even when evaluating protocols with 256 concurrent clients, never rises above 30%.

Multiparty protocols. We also evaluated the SMPC protocols throughput isolated from any other operations. In particular, we evaluate two circuits: *Equality* (EQ) and *GreaterThanOrEqualTo* (GTE). These encrypted circuits are the backbone of the protocols implemented on the prototype as they enable the database to answer the NoSQL API defined in Section 5.2. The

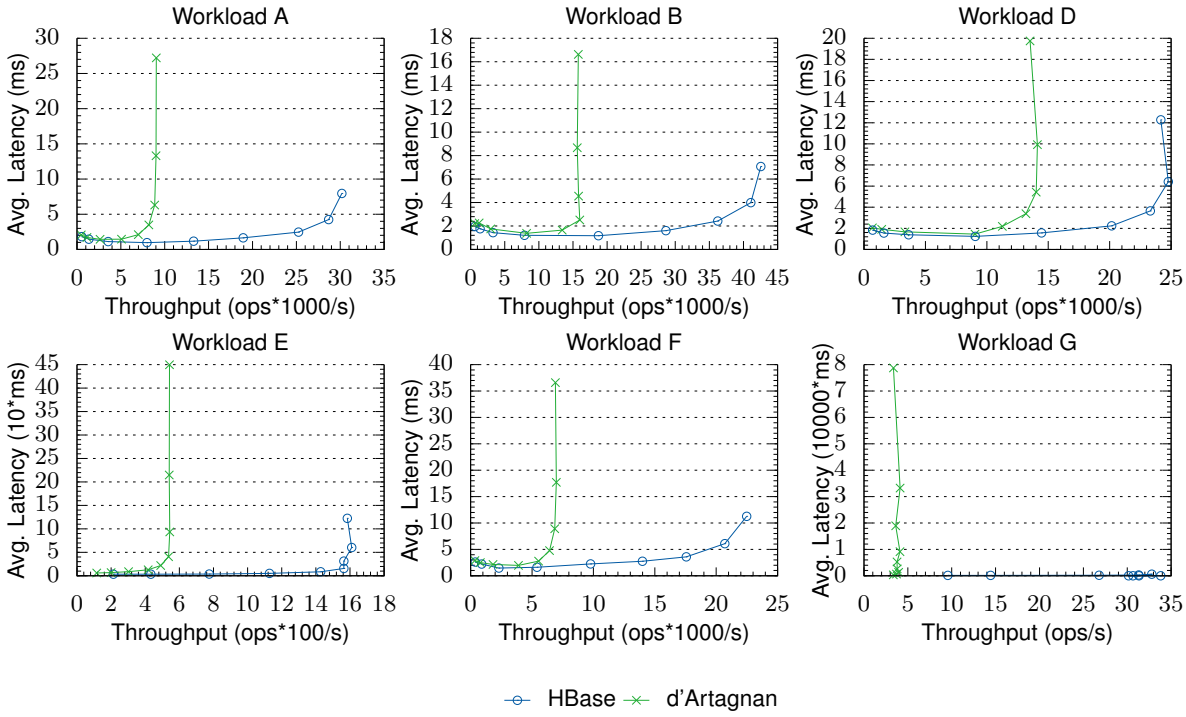


Figure 5.4: d'Artagnan and baseline (HBase) performance with the YCSB Workloads. The benchmarks consider a dataset with 1 Million rows with an increasing number of concurrent clients, from 1 to 256 in a logarithmic scale. The dots ($\times$, $\circ$) in the plot represent an experiment with the increasing number of clients.

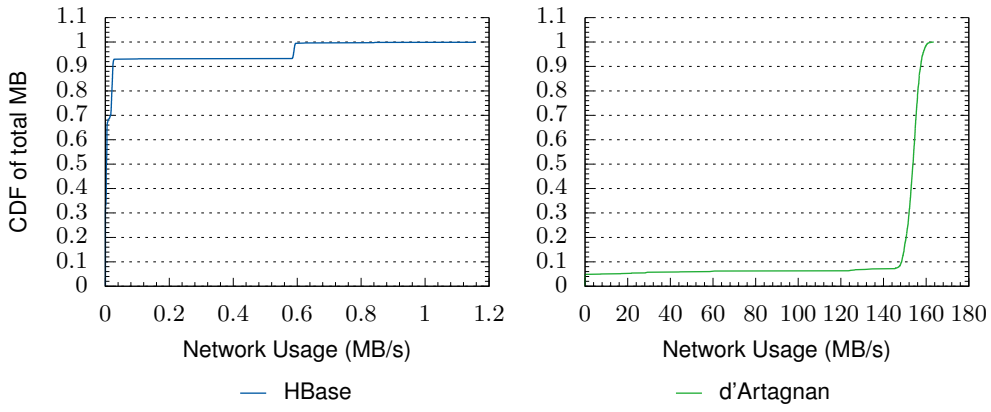


Figure 5.5: Baseline (HBase) and d'Artagnan cumulative distribution function (CDF) of the network usage on workload G with a single client.

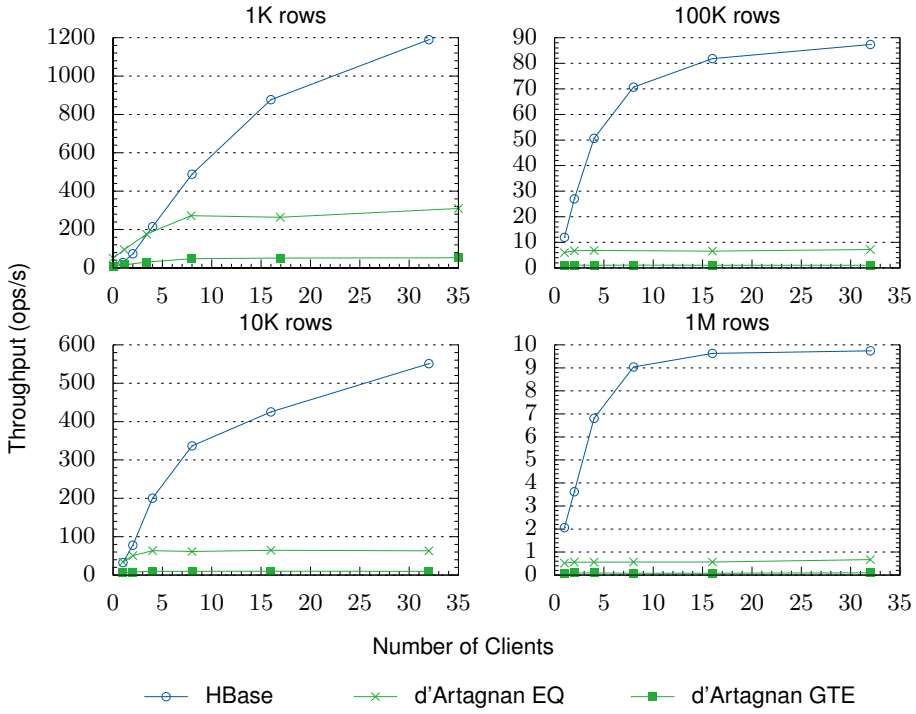


Figure 5.6: d'Artagnan and baseline (HBase) performance on YCSB workloads with 100% Filters. These plots present both systems throughput with an increasing data set and clients (1,2,4,8,16,32).

evaluation measured the protocols performance with different Appointments table size, ranging from 1000 records to 1M. Furthermore, it assesses the protocols scalability for each dataset with an increasing number of clients, from 1 to 32 in a logarithmic scale base 2. The number of clients was restricted to 32 as d'Artagnan prototype reaches a saturation point. The table and request value distributions followed the same approach as the previous evaluation. The baseline was HBase's Equal filter.

Figure 5.6 shows the evaluation results with throughput versus number of clients curves. Overall, both systems throughputs decrease as data size increases, but only the baseline scales with the number clients. For the smaller data set, 1K rows, d'Artagnan prototype EQ protocol peaks at 310 OP/s. In contrast, the baseline has a maximum of 2431 OP/s for the same filter operation. Still on the smaller datasets, the GTE protocol reaches a maximum throughput of 52 OP/s for 32 clients. On the larger datasets, 100 K and 1 M, d'Artagnan prototype has a consistent overhead of 99% compared to the baseline as the system cannot scale with the increasing number of clients. Similar to Workload G, the main bottleneck is the network. On the smallest datasets the network usages ranges on average from 4 MB/s to 90 MB/s as the number of clients increase. After 10 K the bandwidth becomes saturated with just a few clients.

5.6.3 Multi-cloud deployment

A multi-cloud deployment requires a careful analysis of the cloud providers' location and the interconnecting network. The most important aspects are the distance between the third-party infrastructures and their distance to the `Safe Client`. Ideally, the client machine should be a private infrastructure located in the same city as the cloud servers to minimize the requests latency. However, this is not often possible either because the private infrastructure is far from any of the cloud provider servers, as is our case, or because the cloud providers do not have datacenters at the client's geographical area. As such, we present a scenario that illustrates a realistic use-case where deployment on a single city is not available but the `Safe Client` is in a private infrastructure near different cloud providers.

Experimental Set-up. The entire deployment consisted of 4 nodes, three independent HBase servers hosted on Google Cloud, Microsoft Azure and Digital Ocean, and the YCSB benchmark client hosted on Amazon AWS. The nodes were spread out through European countries and were selected to minimize the latency and maximize the available bandwidth. Google's HBase server was located on Frankfurt, Azure's was on Holland and Digital Ocean's was on Belgium. The client machine was also located on Frankfurt. The latency between nodes ranged from 1 ms to 12 ms and the bandwidth from 1 Gbps to 3 Gbps. d'Artagnan's servers were hosted on machines with 4 vCPUs, an SSD Disk and at least 10 GB of main memory. The client machine had 1 vCPU allocated, an SSD Disk and 1 GB of main memory.

This scenario follows a similar approach to the controlled environment but adjusts the appointments table size to 100 K rows. This adjustment was made to simulate a realistic use case where critical data is stored on an untrusted cloud [KTV+18]. As the system's scalability is presented in the controlled environment, this evaluation only considers the YCSB workloads and SMPC protocols for 32 clients. Experiments consists of 3 hour runs.

Figure 5.7 presents d'Artagnan prototype results as the overhead percentage in relation to the baseline, an HBase server on Microsoft Azure. All YCSB workloads follow the same distribution as in the controlled environment. d'Artagnan prototype throughput on workload A and B has a maximum overhead of 95%, peaks at 797 OP/s on workload A while the baseline reaches the 10 KOP/s on workload B. On the workloads D, E and F the overhead is slightly smaller and never surpasses the 43%. On Workload G the prototype peaks at 21.48 OP/s and the baseline at 38.53 OP/s, an overhead of 44%. The EQ protocol has the lowest overhead of 39% in contrast to HBase.

Overall, both experimental settings show that the d'Artagnan main source of overhead are the SMPC protocols. Even though these protocols are among the most efficient in the state-of-the-art the network bandwidth used to evaluate the multiplication gates decreases the overall system's throughput. However, the system's performance is acceptable for privacy-sensitive

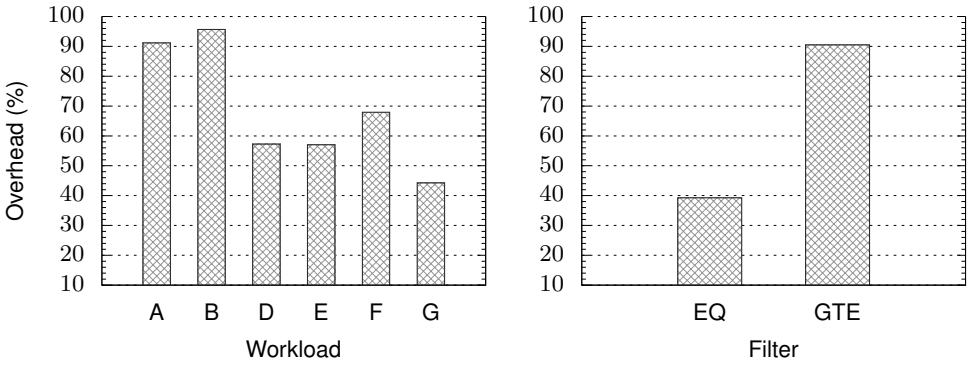


Figure 5.7: d'Artagnan overhead against baseline (HBase) for the YCSB workloads and filters on a multi-cloud deployment for 32 clients and an Appointments table with 100 K rows.

application without real-time performance requirements. In a realistic deployment with cloud providers, the system has an overhead as low as 39%. Furthermore, the current performance is not a hard-limit as novel, ground-breaking SMPC protocols have broken the 1 Billion gates per second barrier [ABF+17] as well as achieved global-scale secure computation [WRK17]. The proposed framework can support additional protocols to tailor the performance for specific application requirements.

5.7 Summary

At a high-level, this Chapter explored a novel approach to outsource range queries to an untrusted third-party. More concretely, we focused on the particular problem of creating a secure key-value NoSQL database on top of untrusted clouds. Whereas existing cryptographic protected databases rely on computational secure cryptographic schemes that centralize information in a single untrusted site, we opt for a decentralized approach that splits data throughout multiple non-colluding parties. d'Artagnan is a novel NoSQL database framework that uses secret sharing as well as SMPC protocols to store and process queries over distributed secrets. With this approach, our system can use a single encryption scheme to process any query without having to necessary disclose partial information and a single data breach does not compromise the user's confidentiality. The main contribution of d'Artagnan is managing the multiple parties, each with an independent database, to create a single logical secure database. Furthermore, its architecture is agnostic to the SMPC protocols and new constructions can be integrated to protected data from an active adversary, support a greater subset of corrupted nodes or increase the system availability.

d'Artagnan's prototype was evaluated with state-of-the-art benchmarks and deployed in market-leading cloud providers. The results show that the secure multiparty protocols protocols currently used by d'Artagnan have a considerable overhead and the main bottleneck is the network bandwidth. Additionally, the cloud evaluation also shows that this approach is feasible

and can even provide a practical alternative for applications that manage a small but highly sensitive dataset (e.g.: medical data, identity management). Applications with large data sets clearly require novel protocols that improve the overall system performance. One possible future research path is exploring SMPC protocols with an offline and online phase. In this model, parties exchange information in the offline phase to evaluate functions in the online phase more efficiently. However, the integration of this model is not straightforward as it is not clear what are the implications in the overall architecture of NoSQL database systems.

Chapter 6

Building oblivious searches from the ground up

In this chapter we depart from the decentralized approach from the previous chapters and instead tackle a common vulnerability of cryptographic protected databases, the access patterns disclosed by search queries. This leakage can be exploited by an adversary to reconstruct the plaintext values of an encrypted database with high accuracy. We address this leakage with CODBS, an oblivious search scheme that redesigns the internal data structures of relational databases to provide a new trade-off between security and performance. More specifically, CODBS generates an access pattern for each query that is indistinguishable from arbitrary accesses and it is more efficient that state-of-the-art constructions (asymptotically and experimentally).

6.1 Introduction

CPD systems often use SSE schemes as they enable a client to store encrypted data in a third-party and evaluate queries remotely over ciphertexts. As discussed in Section 2.2.3, these schemes create an encrypted index that captures the relation between a keyword and a set of documents identifiers, with each identifier pointing to an actual document. The index as well as the documents are encrypted by the client and stored on a remote server. The client can query the index by generating cryptographic tokens for a specific keyword. Given a token as an input, the server can search the index and find the set of documents that contain the queried keyword without having to decrypt any data. However, SSE schemes can disclose confidential information even when proven secure under standard cryptographic assumptions. The main source of leakage are the access patterns revealed by a query, which can lead to statistical analysis attacks [IKK12, CGPR15].

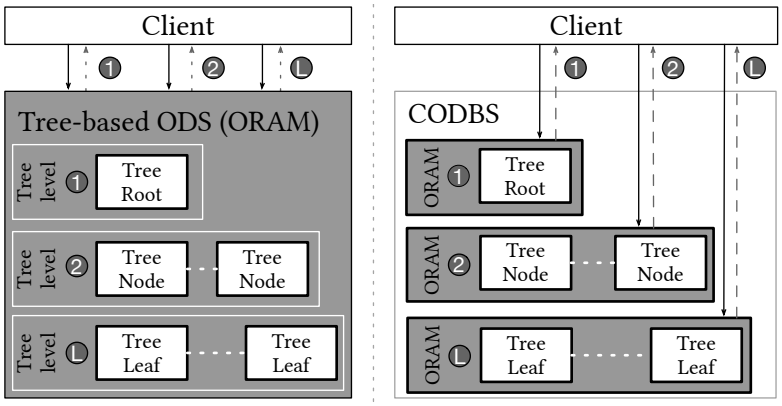


Figure 6.1: Search tree with L levels stored in two different ORAM constructions: a tree-based ODS as prosed by *Wang et al.* [WNL+14] and the CODBS scheme presented in this chapter.

One approach to address the leakage of SSE schemes is to use ORAM [GO96]. With an ORAM scheme, a client can access a remote storage and conceal its access patterns by shuffling and encrypting multiple blocks. This allows the leakage of an SSE scheme to be hidden, by replacing the whole server-side index and document storage with a single monolithic ORAM. However, this approach has a few drawbacks: the SSE client has to keep additional local state (position map and stash); a single query needs multiple communication rounds between the client and the server; ORAM algorithms are bandwidth-intensive even for simple block accesses [SDS+18, PPRY18, SPS14].

The overhead of ORAM schemes can be minimized by using trusted hardware co-located with the encrypted index on the server-side. As presented in Section 2.3.3, an IEE is a trusted hardware technology that offers the possibility to perform arbitrary (verifiable) computations in a clean slate. The internal state of an IEE is assumed to be isolated from other co-located processes, including operating systems and hypervisors. Intel's Software Guard Extensions (SGX) is a prominent instance of an IEE that is widely used to develop novel solutions due to its ubiquity and accessibility in commodity hardware. We will use SGX as a deployment example, even though our approach and security analyses are modular in relation to the trusted hardware security anchors.

Combining ORAM primitives with trusted hardware is a relatively new approach to search over encrypted data with minimal leakage. Existing systems propose novel search algorithms and index data structures optimized for ORAM primitives that effectively lower query latency and the bandwidth used between the client and the server. More concretely, Oblix [MPC+18] uses an oblivious search tree to index the keywords of a database and POSUP [HOJY19] uses an oblivious linked list to store the keywords as well as the documents. In fact, both of these systems improve on the early work of oblivious data structures (ODS) proposed by *Wang et al.* [WNL+14]. Besides these fine-grained SSE systems, there are also full-fledge oblivious database solutions such as Opaque [ZDB+17] and ObliDB [EZ19]. Nonetheless, both systems

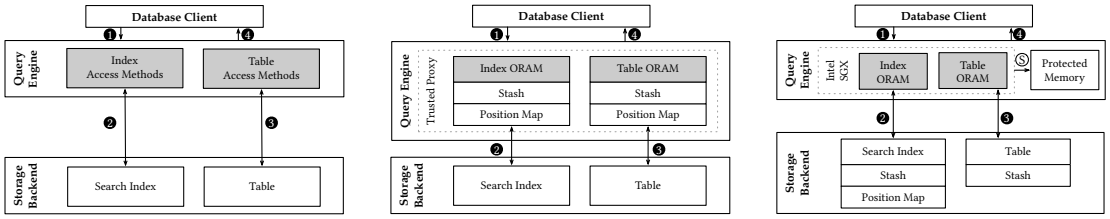
use ORAM algorithms as black-boxes and do not optimize the internal data structures. Trusted hardware is used in these systems as an anchor of trust to search encrypted data on behalf of the client. In Oblix as well as POSUP the enclave is limited to fetch encrypted blocks from the untrusted server and searching for records that match a search key. However, a few systems such as ObliDB take a step further and deploy to an enclave a trimmed-down database engine.

Our main goal in this chapter is to provide a novel oblivious search scheme for relational databases with minimal bandwidth usage and client side state. However, instead of taking the top-down approach of SSE schemes that propose new index data structures and rely on ORAM schemes as black-boxes, we take a bottom-up approach; we design from the ground up an oblivious search scheme that builds on the indexes of existing database systems. This approach ensures that we leverage the extensive research in database systems and that our system design choices are based on realistic and pragmatic assumptions. We do not aim to be fully SQL compliant, but our scheme can be integrated in full-fledge solutions (e.g.: ObliDB) to improve the overall system throughput.

To achieve our goal we present in this chapter a novel oblivious search scheme CODBS inspired by *Wang et al.* tree-based ODS [WNL+14]. Our scheme is deployed on a trusted proxy and uses two independent data structures, one to store a table index and one to store a database table. The search scheme improves over existing work in several key aspects. In comparison to POSUP it does not require auxiliary data structures to keep a relation between keywords and ORAM addresses. Furthermore, our scheme searches over keywords and reduces the position maps to a small constant. Our system is also closely related to Oblix tree-based ODS but it has a lower bandwidth blowup.

More concretely, CODBS is a tree-based oblivious search scheme to store database indexes. This scheme originated from the observation that an oblivious search on a tree-based ODS touches every tree level once in the same order. As such, it is clear that a balanced tree-based ODS only needs to hide which node is accessed in a level and not the level accessed. From this insight, we split the search tree into L smaller ORAM instances, rather than a large ORAM, where L is the search tree height. With this modification, we reduce the bandwidth blowup of *Wang et al.* tree-based ODS from $\mathcal{O}(N \cdot \log(N))$ to $\mathcal{O}(L^2/2 - c)$, here N is the number of data blocks and c is a small constant. This contribution is depicted in Figure 6.1. Additionally, in the original construction the location of tree node can only be reshuffled after a complete tree scan. Our construction can flush a node immediately after an access by generating the position of a node dynamically with a PRF, similarly to Two-ORAM [GMP16]. If the flush is made on a background thread we are able to reduce the query latency by half.

Besides CODBS, we also propose Forest ORAM, an optimized ORAM construction to store database tables. Forest ORAM is used to store the tree levels of CODBS and hide the access patterns. All of algorithms presented in this chapter have been implemented in a complete solution on top of PostgreSQL, one of the most widely used open-source databases. Our



(a) Plaintext Database (b) Naive Oblivious Database (c) Optimized Oblivious Database

Figure 6.2: System models of a plaintext database, a naive oblivious solution and an optimized oblivious system.

implementation is non-intrusive, as it does not require any modification to the database engine. We also measured the average system throughput, latency and resource usage of our solution with an industry-standard benchmark, YCSB [CST+10]. Through a systematic and detailed evaluation we validated the asymptotic improvements of our construction and shown a $\sim 2\times$ to $\sim 4\times$ performance improvement over state-of-the-art constructions that leverage Path ORAM and oblivious data structures [WNL+14, MPC+18].

This chapter is organized as follows, in Section 6.2 we present a high-level model of the the problem addressed in this chapter, the security model and an overview of CODBS. Next, in Section 6.3 we present the security definitions of the encryption schemes used throughout this chapter. The main contribution of this chapter, CODBS, is presented in Section 6.4 and then in Section 6.5 we present Forest ORAM. We prove the security of the proposed scheme in Section 6.6 and then present the experimental system evaluation in Section 6.7. Finally, we summarize the results obtained in this chapter on Section 6.8.

6.2 Problem definition

6.2.1 System model

Databases support a diverse range of data structures and server-side operations to select, filter, aggregate and join data. We focus on minimizing the information disclosed by the *index scan* operator, a fundamental building block of relational databases. By protecting index scans, more complex operations inherit its security guarantees. Our starting point is the typical architecture of a relational database management system, in which a request placed by the client to the remote server is processed in two steps: i. an index scan locates the record of interest in the underlying storage; and ii. an access to the storage retrieves a data block in which the record resides. Database performance hinges on the optimization of accesses to the storage subsystem—in which both indexes and data tables reside—via specialized data structures. Tree-based indexes are used to ensure that most accesses to storage span only a few blocks, and the tree data structures are themselves optimized to minimize cross-block transitions.

To understand the different stages of an index scan, we capture a typical plaintext database system with the model depicted in Figure 6.2a; we consider three main components: a *Database Client*, a *Query Engine* and a *Storage Backend*. In this model, the *Database Client* is a remote application, for instance a web server or a system administrator; it relies on the database management system to store and process data. The actual query processing is handled by the *Query Engine*, which is the most computationally intensive component. However, the *Query Engine* is stateless, and data is stored on block-based data structures. This *Storage Backend* abstracts the storage capabilities available in cloud infrastructures such as elastic file systems or external block storage. Inside the *Storage Backend* there are two data structures, a *Search Index* and a database *Table*. We consider the index a B^+-tree [LY81] that keeps a mapping between a keyword and table record. The database *Table* is a linked list of blocks with each block holding a subset of records.

Our conceptual model is similar to the model of searchable encryption schemes [HOJY19, FBB+18, HOJY19] but has two minor adjustments. First, in SSE schemes the indexes are often a simple mapping (e.g.: Hash table) that have constant access time whereas in our model the index is a B^+-tree that has logarithmic search time. Secondly, our model explicitly handles the accesses to the table (documents in SSE) which is a source of information leakage that is not always included in SSE models. Our goal is to focus on the main components that disclose information in a search query of a standard relational database management system.

During an index scan the components follow a predictable set of steps. A query execution starts with the *Database Client* sending a query to the *Query Engine* (Figure 6.2a-❶). The input query is intercepted by the *Query Engine* which generates a query plan describing the database tables and indexes that must be accessed and the order of the accesses. The *Query Engine* executes an index scan by searching a tree-based index (Figure 6.2a-❷). This index search results in a subset of table pointers that satisfy an input query. For each pointer, the *Query Engine* retrieves its matching table record (Figure 6.2a-❸) and stores it in a result set. The execution flow between *Query Engine* and the *Storage Backend* is repeated until every relevant record is accessed and the complete result set is sent to the *Database Client* (Figure 6.2a-❹).

The execution of a database query has two main types of leakage. The first, commonly addressed with ORAM, is the access patterns revealed by the query engine when accessing the database storage (Figure 6.2a-❷,❸). Every access from the *Query Engine* to the *Storage Backend* consists of either reading or writing a data block in one of the databases' data structures. The sequence of blocks accessed during the tree transversal define a unique path that identifies a small subset of data records. The set of possible results is shortened even further by the identity of the blocks accessed on the table storage, as each table block contains a limited number of database tuples. However, simply hiding the access patterns to the database structures with an ORAM construction is not sufficient. An adversary can learn critical information just from the number of accesses from the table index to the table storage. This information is closely

related with the second leakage considered in our model, the volume leakage. As demonstrated by multiple successful attacks, the size of the result set returned from the database server to the client (Figure 6.2a-❹) and the number of accesses made to a specific record is sufficient to compromise the database [GLMP18, LMP18].

Our approach is based on optimizing an ORAM-based solution to fit the database engine architecture and outsourcing most of the client processing and storage load to a *Trusted Proxy* deployed at an intermediate level of trust. As depicted in Figure 6.2b, in a naive solution the *Trusted Proxy* can be thought of as an interactive oblivious protocol that manages two position-based ORAM constructions and keeps all of the client side state inside the protected environment (stash and position map). One of the ORAM constructions stores the database index, while the other stores the indexed table. We detail the trust model and the optimization approach next.

6.2.2 Trust model

We consider a semi-honest adversary that can observe all communications and computation activity, with the exception of those occurring inside the *client* and *proxy*. Concretely, this implies knowledge of: i.) messages exchanged between client and proxy (Figure 6.2c-❶,❹); ii.) proxy interactions with external memory (Figure 6.2c-Ⓢ); and iii.) proxy interactions with the storage (Figure 6.2c-❸). We assume that client-to-proxy interaction is preceded by a key exchange protocol, to establish a secure channel. This allows our system to rely on standard cryptographic techniques to protect the confidentiality of messages exchanged between client and proxy. Instrumenting IEE-enabled code in this way is a common requirement, and has been shown to be achievable securely with minimal performance overhead [SCF+15]. However, secure channels disclose the size, direction and number of messages. Another relevant issue is related to proxy interactions with external memory. This stems from our deployment setting, where the secure proxy is executing in an IEE-enabled system, physically co-located with adversarial controlled environment. Against these threats, it is expected for the hardware to protect the memory contents [BWK+17, XCP15], but not the access patterns. Our protocol tackles this issue with constant-time implementations [ABB+16, MPC+18]. The leakage that remains are the traffic patterns in the client-to-proxy interface, and the access patterns in the proxy-to-storage interface. The adversary has knowledge of the data blocks in the external storage.

6.2.3 Optimization approach

We now refine the high-level model and detail the system architecture used in this chapter, along with an overview of our optimizations. Our system is a relational database outsourced to a third-party infrastructure, as depicted in Figure 6.2c. The *Trusted Proxy* is hosted in an Intel SGX enclave and we assume that the deployment environment supports the creation of genuine IEEs that can be successfully authenticated with an attestation service. With this approach the *Trusted*

Proxy and *Query Engine* are co-located on the same third-party server, effectively lowering the inter-component latency to a minimum. We are agnostic with respect to the *Storage Backend* but for concreteness require that it provides a standard I/O POSIX interface accessible to the *Trusted Proxy*. The *Query Engine* becomes a passthrough middleware that simply manages client connections, reroutes input client requests to the *Trusted Proxy* and provides an interface to read/write blocks from the database storage. The *Trusted Proxy* also holds an internal secret state which includes secret keys used to encrypt/decrypt blocks from the database storage. We do not detail how this state persists, but we assume it is kept outside of the adversary control and that it can be stored alongside the data blocks, encrypted and sealed with a secret key.

Moving the *Trusted Proxy* to an Intel SGX enclave is challenging as enclaves have a limited pool of protected memory available. Current technology is restricted to 128 MiB but only 93 MiB can actually be used to store and read application data. Even though this limit can be overcome with page swapping, it incurs in additional costly operations, and the issue of access patterns arises again [WAK18]. We address this limitation with CODBS, our novel oblivious search scheme which has multiple composable ORAMs that reduce the client (here proxy-side) storage to a constant factor and allow the protected proxy to function with a small local memory, while guaranteeing leakage-free storage access patterns.

CODBS follows a cascade approach of ORAMs that divides a database B^+-Tree index of height L into L independent ORAMs. Each ORAM stores the nodes of a single tree level and hides any access to an individual node. Similar to *Wang et al.* ODS, the ORAM client (here the *Trusted Proxy*) does not store a position map in memory as the tree nodes keep pointers to its children. Intuitively, a tree search is simply a matter of jumping from level to level and following the pointers to the next ORAM node until a leaf is reached. However, the pointers used in *Wang et al.* ODS are static and can only be updated at the end of tree scan. Our scheme addresses this limitation with dynamic pointers that can be updated immediately after accessing any tree node. Our scheme further reduces the client side memory requirements by keeping the ORAMs stash on the external storage and accessing it with sequential scans. The access to the database *Table* in our scheme is considered as an extension of the tree, i.e., an additional level after the tree leaves.

6.3 Definitions

In this section we present the notation and security definitions used throughout this work. The security parameter is denoted by λ and it is passed as input in unary (i.e., 1^λ). A negligible function in the security parameter is denoted as $\mathrm{negl}(\lambda)$. We consider an adversary $\mathcal{A}$ and a simulator S to be polynomial time algorithms. Our constructions rely on standard notions of a variable-length-input PRF and an IND-CPA symmetric encryption scheme [BR05]. The secret keys are uniformly sampled from $\{0,1\}^\lambda$.

Databases. A plaintext database is a set of data records indexed by a search key $DB = \{(key_1, data_1) \dots (key_n, data_n)\}$. We abstract the search keys as keywords from the set of all finite strings $\mathcal{W} \subseteq \{0,1\}^*$ and the data records $data_i \in \{0,1\}^B$ as binary data blocks of fixed length B. A database query $\tau : \mathcal{W} \to \{0,1\}$ is a predicate that consists of keywords in the domain $\mathcal{W}$ that satisfies a boolean formula. Given an input query τ a database search $DB(\tau) = \{data_i : \tau(key_i) = 1\}$ returns all data records that satisfy the query.

We provide a more detailed definition of the internal data structures that emulate the internal details of relational database. The database keys and data records are stored in a pair of data structures where $\mathcal{I}$ denotes a tree-based index that stores the database search keys and $\mathcal{T}$ denotes a *Table Heap* with the data records. The *Table Heap* is defined as a collection of N table blocks $\mathcal{T} = \{(a_1, data_1), \dots, (a_n, data_n)\}$ associated with a unique address $a_i \in \mathbb{Z}$. The tree-based index $\mathcal{I}$ abstracts the search tree found in databases as a collection of L levels, each one storing multiple tree nodes. The number of nodes grow level by level, with the first level starting with a single root node that points to d (tree fanout) child nodes. The nodes in the following levels subsequently point to new child nodes. This process is repeated recursively until the last level that stores the tree leaves which points to *Table Heap* offsets. Therefore, an index $\mathcal{I}$ is a set of pointers $ptrs_{(l,a)}$ uniquely identified by a pair (l, a) of tree level $l \in \{0, \dots L\}$ and node address $a \in \{0, \dots, d^l\}$. Furthermore a pointer is a set of tuples of search keys paired with a child address $ptrs_{(l,a)} = \{(key_1, p_1), \dots (key_d, p_d)\}$ where the address p_i at level l maps to a block address on the next level $p_i \in \{0, \dots, d^{l+1}\}$. We denote access to the data structures with array notation where a *Table Heap* access returns a table block $data \leftarrow \mathcal{T}[a]$ and a *Table Index* access at level l and address a returns its associated node pointers $ptrs \leftarrow \mathcal{I}[l][a]$. A tree-based index is correct if every node has a single parent.

Ideal storage. We capture the access patterns of a database execution using an idealized storage $\mathcal{M}$ that abstracts an untrusted external storage. This storage is a sequence of K words indexed by a logical address space $[K] = \{1, \dots, K\}$. Each word is a data block of size B that can be individually accessed with a block-based API defined as follows:

- $\mathsf{Alloc}(N) \to \mathcal{D}$: reserves a empty data structure $\mathcal{D}$ that consists of a subset of storage words of size $N \in [0, \dots, K]$ from the storage address pace.
- $\mathsf{Read}(\mathcal{D}, a) \to block$: returns a block of $\mathcal{D}$ at address a.
- $\mathsf{Write}(\mathcal{D}, a, block) \to \mathcal{D}$: updates the data structure $\mathcal{D}$ with a new block on the storage address a.
- $\mathsf{Addrs}() \to \mathcal{X}$: Returns the access patterns trace $\mathcal{X}$ of every API invocation on the ideal storage.

The ideal storage is available to any algorithm at any point of its execution and registers every API request in a public access pattern $\mathcal{X}$. More specifically, an access pattern $\mathcal{X} = \{I_1, \ldots, I_N\}$ is a sequence of instructions $I_i = (op, addr, data)$ where each instruction has an operation defined by the public interface $op \in \{\mathsf{Alloc}, \mathsf{Read}, \mathsf{Write}\}$, a target address within the ideal storage range $addr \in [K]$ and a binary string that is either written to or read from the storage $data \in \{0,1\}^{\mathcal{B}}$. In case of a read instruction the data is initially empty and is filled with the content of the logical address in the storage. In case of an allocation, the requested storage size is specified in the address and the data field is left empty. We denote by $out \leftarrow \mathcal{A}^{\mathcal{M}}(prms)$ the execution of an algorithm $\mathcal{A}$ with access to the ideal storage $\mathcal{M}$ given parameters $prms$. The access pattern of the algorithm execution can be obtained by $\mathcal{X} \leftarrow \mathcal{M}.\mathsf{Addrs}()$.

6.3.1 Oblivious index scan

We define here the syntax and security of an Oblivious Index Scan (OIS), which is realized by our main construction. The goal of this primitive is to leverage specialized ORAMs as building blocks that store the database data structure and ensure that every input query has a constant number of accesses. Intuitively, an OIS scheme starts with an empty data storage, which the *Database Client* fills by outsourcing a plaintext database structure via the *Trusted Proxy* with an initialization algorithm. After this initial step, the *Database Client* establishes a request/response stream with the *Trusted Proxy* such that, for each query sent a pre-defined number of record results is returned (e.g., one record). The complete result set of a query is only returned after multiple requests. This stream-based communication hides the sizes of query results and can be removed if this leakage is not a problem for the application.

In our syntax and security model we omit the stream-based access mechanism, as this introduces unnecessary complexity and because, once it is established that the proposed protocol hides everything except the size of the query results returned to the client, it is straightforward to argue that introducing the stream-based mechanism removes the remaining leakage from client-to-proxy communications. We do analyse the performance trade-offs introduced by this mechanism in Section 6.7.

Definition 1. (Oblivious Index Scan) An oblivious index scan scheme OIS consists of the following two algorithms:

- Init(1^λ, $\mathcal{I}$, $\mathcal{T}$, $prms$)$\rightarrow$(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$): Initialization algorithm that takes as input a *Table Index* $\mathcal{I}$, a *Table Heap* $\mathcal{T}$ and the public database parameters $prms$: (number of blocks N, tree-based index height L, and tree fanout d). The algorithm returns an internal state st, an oblivious search tree $\tilde{\mathcal{I}}$ and an oblivious table $\tilde{\mathcal{T}}$. The oblivious data structures preserve the indexing relation between the input data structures. The internal state is kept securely within the *Trusted Proxy* and it contains the internal state of multiple ORAMs, a secret

key for a symmetric encryption scheme and a secret key of a PRF. The oblivious data structures are stored in the *Storage Backend*.

- Search($st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau$)$\rightarrow$($st', \tilde{\mathcal{I}}', \tilde{\mathcal{T}}', data$): Search algorithm that takes as input the current state st, an oblivious *Table Index* $\tilde{\mathcal{I}}$, a oblivious *Table Heap* $\tilde{\mathcal{T}}$ and an input query τ. The algorithm filters the records that satisfy the query with an oblivious index scan and returns an updated state st', a shuffled oblivious *Table Index* $\tilde{\mathcal{I}}'$, a permuted oblivious *Table Heap* $\tilde{\mathcal{T}}'$ and the resulting $data$ record.

Correctness. An oblivious index scan scheme is correct if for every security parameter λ, every plaintext database DB, every pair of oblivious data structures initialized $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{T}}$ initialized by the Init algorithm and every query τ, the set of the data records of a plaintext database search $DB(\tau)$ with size N is equal to the set of records returned after a sequence of N query searches Search with probability $1 - negl(N)$[1]. As such, correctness is defined by:

$$DB(\tau) = \{\mathsf{Search}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau_0), \ldots, \mathsf{Search}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau_N)\}$$

Security. The security of an OIS construction is defined in the simulation-based real/ideal paradigm. Our security game consists of an adversary that sends a plaintext database of it choosing to the experiment and afterwards sends a sequence of queries. For each query, the adversary receives the access patterns $\mathcal{X}$ of the database to an external storage and a pair of encrypted data structures. In both games, the access pattern includes the addresses of blocks accessed, the instructions (read or write) and the blocks encrypted with a symmetric encryption scheme. We consider an adaptive adversary that can change its attack strategy during the game depending on the access patterns returned by the experiment. Intuitively, an OIS construction is secure if an adversary cannot distinguish if the access patterns were generated by a real-world execution or a simulator that only does arbitrary accesses. In both worlds, the access patterns are captured by an ideal storage $\mathcal{M}$ that is used internally by the ORAM constructions to read/write data blocks. However, in the real-world the ORAM accesses depend on the input queries, while in the ideal world the simulators are only given the databases public parameters, number of blocks, the tree-based index height and fanout. As such, the simulators use the ORAMs as black-boxes to access arbitrary storage addresses. This security definition follows a similar approach to simulation-based definitions of ORAM constructions [PPRY18, AKL+18].

Definition 2. Let OIS = (Init, Search) be an oblivious index scan scheme. For every stateful algorithm $\mathcal{A}$ (the adversary) and S (the simulator), consider the following security game:

[1] This negligible chance of failure matches the probability of failure of a single underlying ORAM scheme. In our proposed construction, the probability is $1 - negl(N) \cdot L$, as L ORAMs are used.

- **Real**$^{OIS}_{\mathcal{A}}(1^\lambda)$: The adversary $\mathcal{A}$ sends the experiment a pair of plaintext database structures $\mathcal{I}$ and $\mathcal{T}$ as well as public parameters: number of *Table Heap* blocks N, *Table Index* height L and fanout d. The experiment initializes a pair of oblivious data structures $(\tilde{\mathcal{T}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{Init}(1^\lambda, \mathcal{I}, \mathcal{T})$ and returns them to the adversary alongside the initialization algorithm access patterns $\mathcal{X}_I$. The adversary follows with a polynomial number of adaptive search queries τ and the experiment outputs updated oblivious structures and the access patterns $\mathcal{X}_S$ of the $\mathsf{Search}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$ function. In the end, the adversary returns a bit b which becomes the experiment result.

- **Ideal**$^{OIS}_{S,\mathcal{A}}(1^\lambda)$: The adversary $\mathcal{A}$ sends to the experiment a pair of plaintext database structures $\mathcal{I}$ and $\mathcal{T}$ and public parameters. The game initializes a pair of dummy oblivious data structures $(\tilde{\mathcal{T}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{S}(1^\lambda, N, L, d)$ and returns them to the client alongside a simulated access pattern $\mathcal{X}_I$. The adversary evaluates a polynomial number of adaptive search queries τ and the experiment returns the updated oblivious structures and simulated access patterns generated by $\mathsf{S}(N, L, d)$. Here the simulator is stateful and the crux is that it does not see the raw data or queries. At the end, the adversary returns a bit b which becomes the experiment result.

OIS is secure if for all $\mathcal{A}$ there exists a simulator S such that:

$$\left| \Pr\left[\textbf{Real}^{OIS}_{\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\textbf{Ideal}^{OIS}_{S,\mathcal{A}}(1^\lambda) = 1\right] \right| < \mathsf{negl}(\lambda)$$

The formal security definition is presented in Section 6.6.1.

6.3.2 Oblivious RAM

We present an extended definition of oblivious RAM schemes that underpins our OIS construction. We follow the classical definition of position-based ORAMs [SDS+18, WNL+14] where a client (e.g.: local machine) remotely accesses data blocks in a server (e.g.: block storage) but modify it in two ways. First, instead of providing a single Access method that reads data from the server, shuffles the blocks and flushes them back, we divide these processes in two distinct functions. Secondly, we explicitly require an external position δ to be passed as input for every oblivious access. Similar to internal position maps, the external position map keeps track of the current location of blocks. However, the external position map also determines the next location where a block must be stored after an oblivious access. As such, the responsibility of correctly book-keeping the location of the blocks is shifted to the ORAM client. We do not restrict the external pmap structure in any way. It depends on the ORAM construction. We note that ORAM without a pmap are also captured if δ is set to empty ($\perp$)

Definition 3. (Oblivious RAM) An oblivious RAM scheme consists of the following three algorithms:

- Build(N)$\rightarrow$(st, $\tilde{\mathcal{D}}$): Initialization algorithm that takes as input a maximum number of blocks N and outputs an internal state st and an initialized data structure $\tilde{\mathcal{D}}$.
- Read((st, δ), $\tilde{\mathcal{D}}$, a)$\rightarrow$(st', $data$): Access operation that takes as input an internal ORAM state st, an external position map δ, the external data structure $\tilde{\mathcal{D}}$ and a block address a. It returns an updated state st' and the external block $data$. This operation does not evict the ORAM internal state nor modifies the external data structure.
- Write((st, δ), $\tilde{\mathcal{D}}$, a, $data$)$\rightarrow$(st', $\tilde{\mathcal{D}}'$): Eviction operation that takes as input an internal state st, an external pmap δ, a data structure $\tilde{\mathcal{D}}$, a block address a and the new block $data$. It evicts stashed blocks, writes $data$ to offset a and returns an updated state st' and data structure $\tilde{\mathcal{D}}'$.

We stress that the division of the Access methods in two operations is only made to clearly expose a position map that can be updated dynamically, as done by our construction in Section 6.4.

Security. In the classical ORAM indistinguishably definition, an ORAM scheme is secure if it generates access patterns independent of the client real accesses. Intuitively, an oblivious access pattern cannot disclose which data the client is accessing, when a data block was last accessed, or if a real access was a read or write operation.

Definition 4. (ORAM security) Let a data request sequence of a client to an external server be denoted by:

$$\vec{y} = ((op_M, a_M, data_M), \ldots, (op_1, a_1, data_1))$$

where M is the sequence size and each op_i denotes a **read**(a_i) or a **write**(a_i, $data_i$) operation where $1 \leq i \leq M$. More specifically, the block read/written is uniquely identified by address a_i, and $data_i$ denotes the data being read/written. Note that request sequences are only sent to the server after the client initializes an ORAM with the Buld algorithm and a **read** or a **write** operation corresponds to the evaluation of the Read algorithm followed by the Write algorithm. Furthermore, the offset 1 corresponds to the most recent operation while the offset M corresponds to the oldest operation. Additionally, Let $\mathcal{X}[\Phi](\vec{y})$ denote the access patterns (possibly randomized) generated by an ORAM construction Φ when accessing a remote storage server. An ORAM scheme Φ is secure if: *1)* for any two data request sequences $\vec{y_1}$ and $\vec{y_2}$ have the same length, their accesses patterns $\mathcal{X}[\Phi](\vec{y_1})$ and $\mathcal{X}[\Phi](\vec{y_2})$ also have the same length and are indistinguishable (computational or statistically); *2)* the data returned on input $\vec{y}$ is correct with probability $>= 1 - negl(|\vec{y}|)$.

We present a security definition that captures ORAM construction with an external position map. Intuitively, an ORAM construction is secure according to Definition 5 if the ORAM client generates the position map independently of the input request sequence.

Definition 5. (External position map ORAM security.) Let $\vec{\delta} = (\delta_M, \ldots, \delta_1)$ denote a sequence of position map states such that δ_i determines the access pattern for request op_i. Additionally, we denote by $\mathcal{X}[\Phi; \vec{\delta}](\vec{y})$ the access patterns (possibly randomized) generated by an ORAM construction Φ when accessing a remote storage server given a sequence of position maps $\vec{\delta_y}$.

Given a data request sequence $\vec{y}$, we say that an ORAM construction with an external pmap $\mathcal{X}[\Phi; \vec{\delta}](\vec{y})$ is secure if it satisfies Definition 4 and there exists an algorithm $\mathcal{O}$ that upon activation outputs a pmap with an identical distribution to that produced by the construction. The algorithm $\mathcal{O}$ generates the pmap without having access to any information besides the database size as a public parameter. Note that $\mathcal{O}$ is not restricted to random algorithms, as an ORAM scheme can be deterministic and simply access the same sequence of addresses (e.g.: full scan of the storage blocks).

6.4 Oblivious cascading scans

Overview. In this section we present our cascading oblivious database search (CODBS) construction. Intuitively, the scheme captures the interaction between the *Trusted Proxy* and the *Storage Backend*. The client starts by issuing an initialization query to the *Trusted Proxy* in order to outsource a local plaintext *Table Heap* and a plaintext *Table Index* to a sequence of $L + 1$ levels of independent ORAMs. Our construction stores the database blocks across each level by following the pattern that emerges naturally from the tree-based indexes in databases such as B^+-trees. As such, every node of a *Table Index* at level l is stored on the ORAM level l. The last ORAM level is reserved for the table blocks. The underlying ORAMs are devoid of an internal pmap and instead we explicitly provide a pmap for every ORAM access. In fact, the locations of the blocks B in an ORAM at level $l + 1$ are stored in its parent node A at level l. As defined in Section 6.3, each plaintext tree node has a list of tree points (key, a) which is enhanced during the initialization process with an additional counter (key, a, ctr). These counters keep the access to the ORAM levels correct and secure.

After the initialization, the client proceeds to issue search queries to the *Trusted Proxy*. With the multiple ORAM levels, a query search consists of cascading from level to level and choosing the next node to access at each step. To gain an intuition on how search proceeds from level to level, consider the following example of an access to a block A at level l. Before moving to a level $l + 1$, the scheme seeks in the block A a pointer (key, a, ctr) to a child node that satisfies its query. If a match is found, the location of the next block to access is calculated with a PRF by providing as input the ORAM level $l + 1$, the address a and the counter ctr. However, before moving to the next level $l + 1$, the counter of the matching pointer is updated and block A is shuffled back in level l to a new position. The new location of A is also calculated using a PRF. This combination of independent ORAM levels with PRFs enable the search to generate the pmap of a block without having to backtrack and shuffle each level without accessing every block

in the *Table Index*. The database search ends by accessing and returning to the client a single *Table Heap* block that satisfies the input query.

CODBS **in detail.** Given a PRF $\mathbf{F} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^\lambda$, an IND-CPA symmetric encryption scheme $\Theta = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ and a position-based ORAM scheme Φ with an external pmap, CODBS is defined by the Init function and the Search function presented in Algorithm 1. In this section we use ORAMs as black-box, but we will later present an optimized construction in Section 6.5. The pmap in CODBS is not a fixed-sized array as in classical position-based ORAM construction. Instead, the location of a block is provided by *location tokens*, i.e.: two outputs sampled from a PRF. The block location is defined by a tuple $(F(m), F(m'))$ containing a token for its current location and a token for its eviction location. These tokens are used by the ORAM scheme to move a block from its original address to a new one after an oblivious access is made. For instance, assuming the underlying ORAM scheme is a construction similar to Path ORAM [SDS+18] the tokens are used to compute uniformly random leaves in the server's binary tree.

Initialization algorithm. The Init algorithm outsources a plaintext database to a pair of oblivious data structures stored in an untrusted server. The goal of this algorithm is to initialize the *Trusted Proxy* internal state and ensure the database is ready to process client queries. The algorithm starts with the generation of secret keys (line 2) and then proceeds to create two additional oblivious structures, an oblivious search tree $\tilde{\mathcal{I}}$ (line 3) and an oblivious table $\tilde{\mathcal{T}}$ (line 4-12). The oblivious search tree $\tilde{\mathcal{I}}$ is the result of the algorithm InitSearchTree. This tree initialization algorithm traverses the plaintext database tree level by level, creates an ORAM for each level l with capacity for d^l blocks and stores the blocks of a tree level in the respective ORAM. The resulting data structure consists of L ORAMs that keep an identical structure to an input tree-based index. Each tree level is assigned to a single ORAM that stores all the of the level's nodes. Before a tree node is written to an ORAM, its internal structure is updated and a unique access counter is added for every pair of (key, ptr). It's important to note that the pointer ptr in a node at level i points to a node offset a at level $i+1$. As blocks are written to an ORAM for the first time, the cryptographic token used by the Φ.Read function starts with a counter set to 0 and the eviction token increments the counter by a single unit. At the end of the index initialization function every parent node can compute the location of its children.

After the index initialization, the Init function creates an additional level to store the *Table Heap* blocks. The initialization process starts with the allocation of an oblivious table $\tilde{\mathcal{T}}$ (line 4) filled with N dummy blocks. Afterwards, the algorithm scans the *Table Heap* block by block (line 5) and generates two cryptographic tokens for each block (line 6-7). The initial location of a block is computed by providing **F** with a unique message composed by the total number of levels $L+1$, the block address a and an initial counter set to 0 (line 6). The eviction token is computed

Algorithm 1: OIS construction.

Function `Init`$(1^\lambda, \mathcal{I}, \mathcal{T}, N, L, d)$
- $\mathsf{sk}_F \leftarrow F.\mathsf{KGen}(1^\lambda)$; $\mathsf{sk}_E \leftarrow \Theta.\mathsf{KGen}(1^\lambda)$
- $(\mathsf{st}_{\tilde{\mathcal{I}}}, \tilde{\mathcal{I}}) \leftarrow \mathsf{InitSearchTree}(\mathcal{I}, \mathsf{L}, \mathsf{d}, \mathsf{sk}_F, \mathsf{sk}_E)$
- $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
- **for** $a \in \{0, \ldots, N\}$ **do**
  - $\delta_t \leftarrow F(\mathsf{sk}_F, L+1||a||0)$
  - $\delta_{t+1} \leftarrow F(\mathsf{sk}_F, L+1||a||1))$
  - $\delta \leftarrow (\delta_t, \delta_{t+1})$
  - $c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
  - $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  - $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c)$
- **end**
- **return** $((\mathsf{sk}_F, \mathsf{sk}_E, 1, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}), (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$

Function `InitSearchTree`$(\mathcal{I}, L, d, \mathsf{sk}_F, \mathsf{sk}_E)$
- $\tilde{\mathcal{I}} \leftarrow []$; $\mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []$
- **for** $l \in \{0, 1, \ldots, L\}$ **do**
  - $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  - **for** $a \in \{0, 1, \ldots, d^l\}$ **do**
    - data $\leftarrow \mathcal{I}[l][a]$
    - data' $\leftarrow []$
    - **for** $i \in \{0, 1, \ldots, |data|\}$ **do**
      - $(key, a') \leftarrow data[i]$
      - $data'[i] \leftarrow (key, a', 1)$
    - **end**
    - $\delta \leftarrow (\mathsf{F}(\mathsf{sk}_F, l||a||0), \mathsf{F}(\mathsf{sk}_F, l||a||1))$
    - c $\leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
    - $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \mathsf{a})$
    - $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \mathsf{a}, \mathsf{c})$
  - **end**
  - $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l$; $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
- **end**
- **return** $(\tilde{\mathcal{I}}, \mathsf{st}_{\tilde{\mathcal{I}}})$

Function `Search`$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$
- $(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}) \leftarrow st$
- a $\leftarrow 0$; ctr $\leftarrow ctr_r$
- **for** $l \in \{0, 1, \ldots, L\}$ **do**
  - $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]$; $\tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  - $\delta_t \leftarrow \mathsf{F}(\mathsf{sk}_F, l||a||ctr)$
  - $\delta_{t+1} \leftarrow \mathsf{F}(\mathsf{sk}_F, l||a||ctr+1)$
  - $\delta \leftarrow (\delta_t, \delta_{t+1})$
  - $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \mathsf{c}) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \mathsf{a})$
  - $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, c)$
  - $(data', a', ctr') \leftarrow \mathsf{Next}(data, \tau))$
  - $c' \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
  - $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, \mathsf{c}')$
  - a $\leftarrow a'$; $ctr \leftarrow ctr'$
  - $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}$; $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
- **end**
- $\delta_t \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||ctr)$
- $\delta_{t+1} \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||ctr+1)$
- $\delta \leftarrow (\delta_t, \delta_{t+1})$
- $(\mathsf{st}'_{\tilde{\mathcal{T}}}, c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
- $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, c)$
- $c' \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
- $(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, \mathsf{c}')$
- $st' \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, ctr_r{+}1, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}})$
- **return** $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$

Function `Next`$(data, \tau)$
- **for** $i \in \{0, 1, \ldots, |data|\}$ **do**
  - $(key, a, c) \leftarrow data[i]$
  - **if** $\mathsf{SelectChild}(key, \tau)$ **then**
    - data[i] $\leftarrow (key, a, c+1)$
    - **return** $(data, a, c)$
  - **end**
- **end**

with a similar message, but the counter is incremented by 1 (line 7). The syntax of the message ensures that each block's location i is independent from previous locations and every other block. During the table scan, every block a is encrypted and stored in the oblivious data structure at a uniform random location defined by its location tokens δ (line 10-11). The function returns the internal state.

Search algorithm. We now describe CODBS' oblivious search algorithm. During this first look at the algorithm we are not concerned with volume leakage and assume that for every input query τ the protocol returns a single *Table Heap* block. This assumption implies that every indexed record is unique and there are no range queries. We address this limitation in Section 6.4.1. With this simplification the algorithm cascades from the first ORAM level to the last, selecting a single node at each level. In detail, a query search has the following steps:

1.) Oblivious tree search. (line 4-15): In this step, the algorithm traverses the L levels of the tree-based index (line 4), fetching a node from each level until it reaches a tree leaf. At every level of the tree scan, the algorithm accesses the tree node at address a stored on an oblivious location defined by the counter ctr. These variables are initially set to the tree root (line 2) and similarly to the initialization algorithm, the current block location tokens are calculated with a PRF (line 6-7). The accessed tree node (line 9) is processed by the Next function which selects a new child node address a' and a counter ctr' to be accessed in the next tree level. This function also returns an updated node $data'$ that is encrypted and evicted to the oblivious data structure (line 12-13). At the end of every level the block pointers, counters and the internal ORAM state are updated (line 14-15).

2.) Node selection. (Function Next): This operation selects a single tree child node address from an input parent node. A tree node is a set of tuples (key, a', c') where each tuple consists of a node address a', a location counter c' and a predicate key. We scan over every tuple (line 2) and check if a tuple key matches an input query τ (line 4). As the choice of which child nodes satisfies an input query depends on the underlying index we abstract this process with the function SelectChild that takes as input a query τ and the current child key. This function returns a boolean result bit b that is set to true if the key satisfies the query. When a child node is found the function increments the counter of the target child node (line 6). This update is made ahead of time, before the child node is accessed, to ensure that the parent node keeps a consistent pointer before it's shuffled back to the ORAM external storage. The function ends by returning the updated accessed node as well as the current location of the next node, defined by the address a and the old counter value c.

3.) Table heap access. (Line 17-23): Finally, after scanning the search tree and reaching a leaf node, the algorithm obtains a single *Table Heap* block pointer. With this information, the current location of the block is calculated with a PRF function (line 17-18) and the block is accessed and evicted (line 20-23). At the end of the algorithm, the *Trusted Proxy* internal state

is updated (line 24) and the resulting block returned to the client.

Multi-User setting. Our protocol mainly considers the single-user setting, but can also be extended to the multi-user setting. We can follow an approach similar to POSUP [HOJY19] and store an access control list (ACL) on the *Trusted Proxy*, as well as a list of user credentials. This meta-data is created by the data owner and outsourced to the remote server during the database initialization process. Given an input query, the *Trusted Proxy* authenticates the request using the user credentials and validates the user's permissions. If the authentication is successful, then the *Trusted Proxy* searches the database obliviously, as defined in Algorithm 1.

6.4.1 Oblivious query stream

Until this point, the Search algorithm was a 1-to-1 function that returned a single table block for every input query. However, the database must support range queries and equality queries that return multiple results. We address this limitation with the insight that any query with multiple resulting records can be unfolded into a sequence of multiple queries with a single result. Additionally, queries can be composed one after the other to obtain an oblivious stream of client requests and database results. Next, we provide a concise description of our solution assuming that the search keys are defined in a continuous domain fully known to the client. This assumption matches existing work in state-of-the-art oblivious data structures [WNL+14] and can easily be dropped at the cost of additional server-side bookkeeping, as is standard in relational databases.

With this observation, the CODBS client is implemented as an algorithm that maintains a constant rate r of requests/responses with the *Trusted Proxy*. The algorithm starts by opening an authenticated channel with the *Trusted Proxy* and proceeds to send queries on a loop at a rate r. The first query starts by searching for the first element in a subrange of the search key domain. The request is processed by *Trusted Proxy* which scans every ORAM level with the Search algorithm. The resulting database block is stashed by the client which keeps sending queries with the consecutive elements in the key domain. A search query ends when the *Trusted Proxy* returns a dummy element that does not satisfy the client query. This query stream is crucial to hide one of the main sources of leakage of search queries over ORAMs, the volume leakage. To address this leakage the query stream remains active by the client, even if there are no new queries to search. In this case, a dummy query is sent to the *Trusted Proxy* and its result is ignored by the client. With this approach, the volume leakage of the system no longer depends on the size of the result set of a query but rather on the rate of requests made to the proxy. This rate is public information that can be adjusted depending on the workload. Regardless of the request rate, the access patterns no longer depend on private data.

6.5 Forest ORAM

We now instantiate the underlying ORAM construction used for each level. A major concern that arises from storing the *Table Heap* in an ORAM is the bandwidth blowup — number of blocks transferred per access — of an oblivious access. Even though *Table Index* size is proportional to the number of blocks in a *Table Heap*, the cost of a database search is dominated by the access to the last level. In our experimental evaluation we verified that there are 50 times more blocks in a *Table Heap* than in a *Table Index* for a small dataset and this difference only increases as the dataset grows. To address this issue, we propose Forest ORAM an extension to Path ORAM that scales with the number of blocks.

Forest ORAM is based on OblivStore [SSS12], a ORAM partition framework that divides a single ORAM construction in multiple, independent P partitions. Each partition is protected with an ORAM scheme and the cost of accessing a single block depends on the number of blocks in a partition and not on the total number of blocks stored on the external storage. However, the initial OblivStore's constructions was proposed before tree-based constructions became standard and thus uses a hierarchical ORAM scheme for each partition, resulting in a worst-case bandwidth blowup of $\mathcal{O}\sqrt{N}$. In Forest ORAM we replace the hierarchical framework with Path ORAM and optimize the number of partitions to lower the bandwidth overhead of accessing a single Path ORAM and prevent a partition from overflowing. Forest ORAM has $\mathcal{O}(\log(N) - \log(P))$ bandwidth blowup and a $\mathcal{O}(\log(P))$ upper bounded stash.

The security of OblivStore, as with most tree-based ORAM algorithms, is based on the assumption that the stash is stored securely by the client. However, if the stash is stored inside an IEE's secure memory, every access to the stash is disclosed to the adversary. Specifically, an adversary can learn the stash offset accessed and track the movement of blocks within the stash. This leakage can result in linkability attacks [SSS12]. This problem also affects OblivStore as it keeps an individual stash for each partition. The issue becomes clear with the following example; after an oblivious access, OblivStore moves an accessed block from a partition A to a partition B. Without any modification to the original construction, the movement between partitions is simply a matter of removing a block stored in partition A and writing it to the partition B stash. However, if the adversary can trace this exchange of blocks, it learns the exact location where the block was stored. This information would have never been disclosed on a classical client-server deployment, which compromises OblivStore's security.

We addresses this issue with a single oblivious stash shared between all of the partitions. This oblivious stash stores blocks from every partition and has a fixed size equal to the upper bound of excepted blocks, i.e., $\mathcal{O}(\log(P))$ blocks. We denote the oblivious stash as $\mathtt{OS}$ and we use the standard set notation to denote any access to the stash. To ensure a uniform access pattern, all operations to the stash implicitly touch every element and any conditional logic, such as if conditions, or assignments are executed with constant time operators. We also considered

that the Forest ORAM algorithms executed inside an IEE are implemented with constant time guarantees to prevent any access patterns from being disclosed trough side-channel attacks. However, we refrain from explicitly presenting low-level detailed algorithms with constant time operators as there are several standard approaches that can be applied [MPC+18, HOJY19, ABB+16].

The Forest ORAM construction is defined by the algorithms presented in Algorithm 2. Whereas state-of-the-art position based ORAM algorithms provide a single Access method, we split this method in two main functions Read and Write. Furthermore, we also explicitly define the initialization algorithm Build that allocates an external memory structure and setups the construction parameters. Our algorithms are mostly similar to Path ORAM definition [SDS+18] and we explicitly highlight our modifications in blue, which mostly include the division of blocks in multiple partions as done by OblivStore.

Intuition. We first start with an intuitive overview of the protocol. Following OblivStore's partition framework, the untrusted server storage is divided into P individual partitions. The N outsourced blocks are uniformly distributed between the partitions, with each partition storing about N/P blocks each. A partition stores the data in a binary tree, similar to Path ORAM. The nodes in the binary tree are known as buckets and each one stores up to Z data blocks.

Main Invariant. Forest ORAM has one main invariant. Every block is mapped to a uniformly random partition p and a uniformly random leaf l in a partition tree. If a block a is mapped to a partition p and a leaf l, the block can either be found on a client cache or on a bucket at partition p in the path from the root to the leaf l.

To access (read or write request) a block from the server, Forest ORAM uses the external position map to obtain the location of the block. Given a partition p and a leaf l, the entire path from the tree root in partition p to the tree leaf l is transferred to the client stash. The client shuffles the blocks and flushes a new set of blocks to the path accessed. To prevent any number of subsequent accesses to the same block from being disclosed to an adversary, blocks are re-assigned to new locations after each access.

When a block is re-assigned, its partition as well as its tree leaf are sampled from a uniform random distribution. The blocks stay on the client-side stash until a client request can flush it back to the external storage. Similar to OblivStore, we prevent the stash from overflowing with a periodic eviction process that flushes blocks from the stash to the partitions. This process is independent of the client requests and the client cache size.

External position map. The external position keeps track of the blocks locations and is managed by an application using Forest ORAM. Intuitively, this position map could be as simple

as an inverted index that maps a block address a to a tuple with the block location (x, y) where x denotes a tree leaf and y a partition. Additionally, the position map is updated after every access to move blocks to new uniform random position. In Forest ORAM we abstract the details of the implementation of the external position map and define it as a tuple of location tokens $\delta = (\delta_t, \delta_{t+1})$. Each location token is a bitstring $\{0,1\}^D$ with size D. Every Forest ORAM request (Read/Write) on address a receives a location token where δ_t is the current location of a and δ_{t_1} is a new location of a after an access. We denote by $\tau : \delta \rightarrow (x, y)$ a function that takes as input a location token and returns its associated coordinates, leaf x and partition y.

Server Storage. The server storage is divided into P partitions of Path ORAMs such that $P = \log(N)$. Each partition contains $2^{\log(N/P)+1} - 1$ blocks of fixed size B. The partitions are independent Path ORAM constructions with a binary tree of height $L = \{0, 1, \ldots, 2^{\log(N/P)}\}$ and each tree node is a bucket containing Z data blocks. The blocks are structured as a tuple (δ, a, $isDummy$, $data$) such that each block contains its $data$, a bit $isDummy$ that defines if the $data$ is real or free with dummy data, the real block offset a and its current location token δ. It's important to note that we assume that blocks are implicitly encrypted before being stored on the external storage to hide its information from an adversary.

Path. Consider $x \in \{0, 1, \ldots, 2^{L-1}\}$ a leaf node in a tree on partition $y \in \{0, 1, \ldots, P\}$. Each leaf node x in y defines a unique tree path that starts on the tree root and ends on the leaf. We denote by $\mathcal{P}(x, y)$ the set of buckets in partition y on the unique path defined by x. We further denote by $\mathcal{P}(x, y, l)$ a bucket at level l in $\mathcal{P}(x, y)$. Finally, we denote by $\mathcal{L}(y, l)$ the set of all the buckets in partition y at tree level l.

Client Storage. The client's storage consists of a single oblivious stash `OS` shared between every partition. The stash is a temporary holding place for blocks accessed on a partition that have not been evicted to the server. A subset of the stashed blocks is flushed after a partition access, while the remaining blocks are evicted by a background eviction process. As proven in OblivStore [SS13] the size of the stash is upper-bounded by the number of partitions $\mathcal{O}(\log N)$.

Initialization. Forest ORAM is initialized by the Build function defined in Algorithm 2. This function allocates an external memory structure $\tilde{\mathcal{D}}$ for P partitions, each with a Path ORAM tree with height L and Z blocks. The allocated blocks are overwritten by dummy data blocks. The algorithm returns the client state with the internal parameters and an empty stash `OS`.

In Forest ORAM, a block access to an address a is done with a Read request followed by a Write request. First, the client downloads a block from the server with the function Read, updates the block in case of a write operation and flushes the updated block back to the server with the Write function. The functions are described by the following two steps:

Algorithm 2: Forest ORAM

```
1  Function Build(N)
2      P ← log(N) L ← log(N/P)
3      D̃ ← M.Alloc(P · 2^L · Z)
4      Bucket ← []
5      for b ∈ {0, 1, ..., Z} do
6          Bucket[b] ← ({0}*, {0}*, 1, {0}*)
7      end
8      for i ∈ {0, 1, ... P · 2^L} do
9          M.Write(D̃, i, Bucket)
10     end
11     return (([], P, L), D̃)

1  Function Read(st, D̃, a)
2      ((S, P, L), δ) ← st; (δ_t, _) ← δ
3      (x, y) ← τ(δ_t)
4      for l ∈ {0, 1, ..., L} do
5          S ← S ∪ M.Read(D̃, P(x, y, l))
6      end
7      data ← Read block a from S
8      return ((S,P,L), data)

1  Function Write(st, D̃, a, data*)
2      ((S, P, L), δ) ← st; (δ_t, δ_{t+1}) ← δ
3      (x, y) ← τ(δ_t)
4      (x_{t+1}, y_{t+1}) ← τ(δ_{t+1})
5      S ← (S − {(δ, a, 0, data)}) ∪ {(δ_{t+1}, a, 0, data*)}
6      for l ∈ {L, L − 1, ..., 0} do
7          S' ← {(δ', a', 0, data) ∈ S :
             P(x, y, l) = P((τ(δ'), l)}
8          S' ← Select min(|S'|, Z) blocks from S'.
9          S ← S − S'
10         D̃' ← M.Write(D̃, P(x,y,l), S')
11     end
12     return ((S,P,L), D̃')
```

1.) Block Access (Function Read**):** The function generates the location of offset a from the location token (line 3), reads from the remote server the blocks in path $\mathcal{P}(x, y, l)$ defined by the leaf x, partition y and tree level l (line 4-6). The blocks are stored on the stash and the requested block is returned to the client (line 7-8).

2.) Block Flush (Function Write**):** The function generates the current location of offset a as well as its new position from the location's tokens (line 3-4). The stash is updated with new block $data^*$ and the new location of address δ_{t+1} (line 5). The block eviction process (line 6-11) scans a tree path level by level, from the leaf to the root. At each level, the function selects blocks from the stash with a path $\mathcal{P}(\tau(\delta'), l)$ that intercepts the path $\mathcal{P}(x, y, l)$ accessed on the Read function. The selected blocks are trimmed to limit the tree node's capacity to Z (line 8). Finally, the resulting blocks are removed from the stash (line 9) and evicted to the external storage (line 9).

6.5.1 Background eviction

A core component of OblivStore's construction is the background eviction process that prevents the client-side stash from overflowing. However, the algorithm used by OblivStore's is not applicable to Forest ORAM as it is bounded to hierarchical ORAMs. Furthermore, simply evicting blocks from the stash after an oblivious access can disclose the new location of block. We present a new eviction algorithm that addresses the two main challenges of a background eviction:

Algorithm 3: Background eviction.

1 **Function** `Evict`$(st, \tilde{D}, a)$
2 (S, P, L) $\leftarrow$ st
3 $y \leftarrow$ Random$(1, \ldots, P)$
4 **for** $l \in \{0, 1, \ldots, L\}$ **do**
5 $S \leftarrow S \cup \mathcal{M}$.**Read**$(\mathcal{L}(y, l))$
6 **for** $x \in \{0, \ldots, 2^{(l+1)} - 1\}$ **do**
7 $S' \leftarrow \{(\delta, a, 0, data) \in S : \mathcal{P}(x, y, l) = \mathcal{P}(\tau(\delta), l)\}$
8 $S' \leftarrow$ Select min($|S'|$, $Z * 2^{(l)} - 1$) blocks from S'.
9 $S \leftarrow S - S'$
10 **end**
11 $\mathcal{M}$.**WriteBucket**$(\mathcal{L}(y, l), S')$
12 **end**

Minimizing stash size. The background process runs at an eviction rate $c \cdot v$, such that $c > 0$ is the eviction rate and v is the clients request rate, meaning that for every oblivious access there are c background evictions executed. The eviction process maximizes the number of real blocks in a partition and attempts to replace every dummy block with a real stashed block.

Oblivious access pattern. The access patterns of the partitions selected for eviction have to be independent of the data access patterns of client requests and from the stashed blocks. Furthermore, a partition must always be evicted if chosen, even when there are no stashed blocks to evict.

The eviction background process is presented in Algorithm 3. The protocol resembles the Path ORAM eviction function but has a few differences to ensure that blocks are written to partitions independently of the data access patterns. The algorithm transverses the binary tree level by level, from the root to the leaf, reading every block and flushing stashed blocks to the server. Conceptually, Forest ORAM Write eviction is a vertical operation on a tree branch while the background eviction is an horizontal process. We slightly abuse the notation of $\mathcal{M}$.Read and $\mathcal{M}$.Write to denote that every node in tree level is read/write from/to the server.

The algorithm is described by the following 4 steps: **1.)** (line 3) choose a random partition y to evict; **2.)** (line 4-5) Start to iterate the tree level by level $\mathcal{L}(y, l)$, and at every level read the nodes from the server on to the stash; **3.)** (line 6-10) Choose from the stash the set of blocks that can be stored on the current level. This selection filters from the stash the blocks with a path that belong to the selected partition and can be written to the current level; **4.)** (line 11) Evict the batch of selected blocks S' to the current tree level. At each tree level l, blocks are written with a deterministic access pattern from the first node to the last $\{0, 1, \ldots, 2^l\}$.

6.5.2 Asymptotic analysis

The Forest ORAM bandwidth blowup per access is similar to Path ORAM, requiring $2 \cdot Z \log(N)$ blocks to be transferred from the server to the client. The major difference is the partition of the tree in multiple subtrees, each with only $\log(N/P)$ blocks leading to a total bandwidth blowup of $2 \cdot Z(\log N - \log P)$.

In a CODBS search, we can further decrease the online bandwidth by pushing some I/O to a background process. The CODBS search algorithm, as defined in Algorithm 1, accesses the levels of a *Table Index* with a sequence of Read and Write functions, in this same order. However, note that each level is an independent ORAM and that the levels grow by a factor of 2 from the roots to the leaf. As such, an oblivious access to level i has a lower bandwidth blowup than an access to level $i+1$. Furthermore, consider an execution model where a main thread processes the CODBS search and an additional background thread can evaluate any function. If we evaluate the function Write (Algorithm 3) in the background thread, the main thread can fetch a block from level i and move on to access the next level $i+1$. The eviciton of the block in level i is done by the background thread which flushes the blocks before the main thread fetches all blocks from level $i+1$. By induction, we can apply this process to every level and the CODBS search only processes Read functions, effectively decreasing the only bandwidth blowup to $Z \log(\log N - \log P)$.

6.6 Security analysis

CODBS is designed to leak only the maximum number of blocks stored in the *Table Heap* and the *Table Index*. Intuitively, from the perspective of an adversary, the database is a black box that takes as input two public parameters: the database size and a query sequence. The database processes the queries and outputs an access pattern that only depends on the database size and not on the database contents or input queries. As such, an adversary cannot learn from the access patterns or the storage layout of the database any additional information that compromises the database content's such as the order or range between records. Besides the query access pattern leakage our construction also tackles an additional issue, the query volume leakage. Overall, our construction captures a security notion stronger than state-of-the-art searchable encryption schemes and encrypted databases but slightly weaker than a full server-side oblivious database without any leakage.

The security analysis of CODBS is almost straightforward from the composition of black-box position based ORAMs. We focus on the Search algorithm as the same arguments are applicable to the Init algorithm. For every input query, the adversary observes a fixed number of blocks accessed that depend on the public parameter of the *Table Index* height L. Furthermore, each level access generates an external access pattern that is independent of the input query

and the database's data. To an adversary, the external access patterns are indistinguishable from an arbitrary sequence of accesses as the accessed locations are the result of an ORAM construction.

CODBS relies on an IEE to establish a secure perimeter with a small but protected memory region, which we instantiate with Intel SGX. Even though we assume that an SGX enclave ensures the confidentiality and integrity of the memory pages, the access to each individual page is still disclosed to an adversary. In fact, an adversary may attempt to compromise the security of CODBS by exploiting the memory access patterns during its execution in an IEE. To address this vulnerability in CODBS, the algorithms presented in Section 6.4 and Section 6.5 have to be modified to remove any conditional logic and instead use operators that provide constant-time execution. However, we choose to present the algorithms without constant-time properties to provide the reader with a more intuitive description. Nonetheless, a constant-time implementation of these algorithms can be achieved by using standard operators in the literature such as oblivious assignment and oblivious equality comparison operators [HOJY19, MPC+18, ABB+16]. Furthermore, we also assume that the PRF, authenticated encryption scheme and the function SelectChild are constant-time.

Finally, we address the query volume leakage with a continuous stream of fixed sized requests/responses. The stream is continuously generated by the database client which controls the rate of queries sent to the database. For every input query, the database returns a single database result with a fixed public size (e.g: size of a *Table Heap* block) to prevent an adversary from distinguishing between dummy results and real results. Furthermore, as the number of requests/responses exchanged depends on the request rate and not on the size of the query results, the generated stream is indistinguishable from any arbitrary sequence of requests and responses. As such, an adversary cannot disclose how many real queries were issued, the order in which queries are sent or which results corresponds to each query. The composition of an oblivious query stream with the oblivious *Trusted Proxy* access patterns ensures that an adversary cannot distinguish between a real database execution from an arbitrary sequence of accesses.

6.6.1 Security model

We now present our formal game-based security model and the security proof of the CODBS protocol. We present in Figure 6.3 the game-based security definition of an oblivious index (Definition 2). During the execution of the game, an ideal storage $\mathcal{M}$ is available to ORAM primitives in both worlds. In the real game $\mathbf{Real}_{\mathcal{A}}^{OIS}(1^{\lambda})$ the adversary $\mathcal{A}$ starts by interacting with the CODBS protocol by providing a plaintext database to be encrypted and stored on oblivious data structures. The adversary is given the complete sequence of accesses to the external storage $\mathcal{M}$ after the initialization algorithm and the pair of resulting data structures.

$\textbf{Real}_{\mathcal{A}}^{OIS}(1^\lambda)$	$\textbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda)$
$(\mathcal{I}, \mathcal{T}, N, L, d, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$	$(\mathcal{I}, \mathcal{T}, N, L, d, st_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\lambda)$
$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}) \leftarrow Init^{\mathcal{M}}(1^\lambda, \mathcal{I}, \mathcal{T}, N, L, d)$	$(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}) \leftarrow Sim_{Init}^{\mathcal{M}}(1^\lambda, N, L, d)$
$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$	$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$
$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_2(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$	$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_2(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$
while $\tau \neq \perp$	**while** $\tau \neq \perp$
$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, d) \leftarrow Search^{M}(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$	$(\tilde{\mathcal{I}}, \tilde{\mathcal{T}}, d) \leftarrow Sim_{Search}^{\mathcal{M}}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$
$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$	$\mathcal{X} \leftarrow \mathcal{M}.Addrs()$
$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_3(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$	$(\tau, st_{\mathcal{A}}) \leftarrow \mathcal{A}_3(st_{\mathcal{A}}, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \mathcal{X})$

Figure 6.3: OIS Real and Ideal game definition

The adversary follows by adaptively requesting new search queries and for each one receives the respective access patterns. In the ideal game $\textbf{Ideal}_{S,\mathcal{A}}^{OIS}(1^\lambda)$, the interaction between the adversary and the simulator S follows the same flow of interactions. However, the sequence of access patterns and oblivious data structures generated by the simulator does not take into account the adversary plaintext database or its queries. Instead, the access patterns are generated from public parameters.

Our security model focuses on a passive adversary that can observe every interaction between the *Trusted Proxy* and the external database storage. However, the adversary cannot create arbitrary instances of the protocol, or forge requests and data blocks. In fact, our proof can be extended to an active adversary in the IEE model [BPSW16] (Section 2.3.3) albeit at the cost of a more extensive proof that would detract from the main concern of our construction: the external database access patterns. Nonetheless, we provide a brief description on the existing mechanisms that can be used to make CODBS secure against active attackers. Intuitively, our construction can be loaded in an IEE with an attested key exchange protocol that ensures that only a single instance of the protocol can establish a secret key between the database client and the *Trusted Proxy*. Furthermore, a secure communication channel between the client and the database prevents the adversary from providing valid inputs to either party. Finally, an authenticated encryption scheme and a sequence of numbers can be used to ensure that the database blocks in the external storage cannot be forged.

6.6.2 Game-based proof

Let $\Phi = (Build, Read, Write)$ be a constant-time position based ORAM with the security guarantees in Definition 6.3.2. Furthermore, let **F** be a PRF with domain D and output Range R with prf-security as defined by *Bellare and Rogway* [BR05] (Section 2.1.2). Furthermore. let $\Theta = (Gen, Enc, Dec)$ be an IND-CPA encryption scheme according to *Shoup and Boneh* [BS17] (Section 2.1.1). CODBS is secure according to Definition 6.3.1.

Init($\mathcal{I}$, $\mathcal{T}$, N, L, d)

$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); \mathsf{sk}_E \leftarrow \Theta.Gen(1^\lambda)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots, d^l\}$ **do**
    $\delta \leftarrow (\mathsf{F}(sk_F, l||a||0), \mathsf{F}(sk_F, l||a||1))$
    $d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$
    $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, d')$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, \ldots, N\}$ **do**
  $\delta_c \leftarrow \mathsf{F}(sk_F, L+1||a||0)$
  $\delta_n \leftarrow \mathsf{F}(sk_F, L+1||a||1)$
  $\delta \leftarrow (\delta_c, \delta_n)$
  $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$
$st \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, 1, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}})$
return $(st, (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$

Search(st, $\tilde{\mathcal{I}}$, $\tilde{\mathcal{T}}$, τ)

$(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}) \leftarrow st$
$a \leftarrow 0; c \leftarrow ctr_r$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (\mathsf{F}(\mathsf{sk}_F, l||a||c), \mathsf{F}(sk_F, l||a||c+1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
  $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
  $o_R \leftarrow \mathsf{Next}(data, \tau)$
  $(data', a', c') \leftarrow o_R$
  $data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
  $o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
  $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W$
  $a \leftarrow a'; c \leftarrow c'$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta_c \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||c)$
$\delta_n \leftarrow \mathsf{F}(\mathsf{sk}_F, L+1||a||c+1)$
$\delta \leftarrow (\delta_c, \delta_n)$
$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
$st' \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, c+1, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}})$
return $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$

InitNode(Node)

$d' \leftarrow []$
for $i \in \{0, 1, \ldots, |Node|\}$ **do**
  $(key, a') \leftarrow Node[i]$
  $d'[i] \leftarrow (key, a', 1)$
return d'

Figure 6.4: Extended real game.

Sim$_{Init}(N, L, d)$

$\mathsf{sk}_E \leftarrow \Theta.Gen(1^\lambda) : g \leftarrow\$\ Func(D, R)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c \leftarrow 0$
for $l \in \{0, 1, \ldots L\}$
  $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \tilde{\mathcal{I}}') \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots d^l\}$
    $\delta \leftarrow (g(c), g(c+1))$
    $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}'}, \delta), \tilde{\mathcal{I}}', \{0\}^*)$
    $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}'}, \delta), \tilde{\mathcal{I}}', \{0\}^*, data)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}'}, \tilde{\mathcal{I}}') \leftarrow o; c \leftarrow c + 2;$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \tilde{\mathcal{I}}';$
$\tilde{\mathcal{T}}(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, 1, \ldots, N\}$
  $\delta \leftarrow (g(c), g(c+1))$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$
  $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, data)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow o$
$\mathsf{st}_{Sim} \leftarrow (N, L, d, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathsf{st}_{\tilde{\mathcal{T}}}, g, c + 2)$
return $(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

Sim$_{Search}(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

$(N, L, d, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathsf{st}_{\tilde{\mathcal{T}}}, g, c) \leftarrow \mathsf{st}_{Sim}$
for $l \in \{0, 1, \ldots L\}$
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (g(c), g(c+1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*)$
  $data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, \{0\}^*, data)$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o; c \leftarrow c + 2$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}'_{\tilde{\mathcal{I}}_l}$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta \leftarrow (g(c), g(c+1))$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*)$
$data \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
$o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, \{0\}^*, data)$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow o$
$\mathsf{st}_{Sim} \leftarrow (N, L, d, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathsf{st}_{\tilde{\mathcal{T}}}, g, c + 2)$
return $(\tilde{\mathcal{I}}, \tilde{\mathcal{T}})$

Figure 6.5: OIS ideal simulators.

Theorem 6.6.1. The CODBS construction defined by Algorithm 1 is a secure Oblivious Index Scan according to Definition 1 if Φ is an Oblivious RAM scheme, Θ is an IND-CPA symmetric encryption scheme and F is a PRF.

The security proof of Theorem 6.6.1 is described through a sequence of 5 games presented from Figure 6.6 to Figure 6.9. We denote by G_i the i-th game and by $Pr[G_i = 1]$ the probability that Game i outputs 1. Each game is a transition from the real game defined in Figure 6.4 with a slight modifications until the last game that is identical to the ideal game with the simulators defined in Figure 6.5.

Game G_0**.** G_0 is defined by the real security game **Real**$^{OIS}_{\mathcal{A}}(1^\lambda)$ (Figure 6.3) instantiated with the CODBS construction which results in the extended real game presented in Figure 6.4. This extension inlines the InitSearchTree function and encapsulates the addition of a child counter to the tree nodes in the function InitNode.

$$Pr[\textbf{Real}^{OIS}_{\mathcal{A}}(1^\lambda) = 1] = Pr[G_0 = 1]$$

Game G_1**.** The following game G_1, presented in Figure 6.6, makes two modifications on the real world game, which simplify the next steps. First, it adds a global counter to the protocols

Init$(\mathcal{I}, \mathcal{T}, N, L, d)$

$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); \mathsf{sk}_E \leftarrow \Theta.Gen(1^\lambda)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c_S \leftarrow 0; \mathcal{I}_A \leftarrow [][]$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots, d^l\}$ **do**
    $\delta \leftarrow (\mathsf{F}(sk_F, l||a||0), \mathsf{F}(sk_F, l||a||1))$
    $d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$
    $\mathcal{I}_A[l][a] \leftarrow d'$
    $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, d')$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o; c_S \leftarrow c_S + 2$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, \ldots, N\}$ **do**
  $\delta_c \leftarrow \mathsf{F}(sk_F, L+1||a||0)$
  $\delta_n \leftarrow \mathsf{F}(sk_F, L+1||a||1)$
  $\delta \leftarrow (\delta_c, \delta_n)$
  $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[a])$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$
  $c_S \leftarrow c_S + 2$
$st \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, c_S, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathcal{I}_A)$
return $(st, (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$

Search$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$

$(\mathsf{sk}_F, \mathsf{sk}_E, ctr_r, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S, \mathcal{I}_A) \leftarrow st$
$a \leftarrow 0; c \leftarrow ctr_r$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (\mathsf{F}_{\mathsf{sk}}(l||a||c), \mathsf{F}_{\mathsf{sk}}(l||a||c+1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
  $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
  $o_R \leftarrow \mathsf{Next}(\mathcal{I}_A[l][a], \tau)$
  $(data', a', c') \leftarrow o_R$
  $data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
  $o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
  $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W$
  $a \leftarrow a'; c \leftarrow c'$
  $c_S \leftarrow c_S + 2$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta_c \leftarrow \mathsf{F}_{\mathsf{sk}}(L+1||a||c)$
$\delta_n \leftarrow \mathsf{F}_{\mathsf{sk}}(L+1||a||c+1)$
$\delta \leftarrow (\delta_c, \delta_n)$
$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
$st' \leftarrow (\mathsf{sk}_F, \mathsf{sk}_E, ctr_r + 1, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S + 2, \mathcal{I}_A)$
return $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$

InitNode(Node)

$d' \leftarrow []$
for $i \in \{0, 1, \ldots, |Node|\}$ **do**
  $(key, a') \leftarrow Node[i]$
  $d'[i] \leftarrow (key, a', 1)$
return d'

Figure 6.6: Game 1 hop.

internal state. The counter does not modify the protocol but provides a unique, non-repeatable value. The counter is incremented twice after every pair of Φ.Read/Φ.Write function, i.e., when the initialization protocol stores a database block in on the external data structures and after a block is accessed during a query search. The second modification is the addition of an ideal structure $\mathcal{I}_A$ that stores the tree-based nodes generated by InitNode. The data stored on this structure is identical to the data blocks stored on the L ORAM levels and does not modify it in any way. As this modification does not change the game execution it is clear that the adversary gains no additional advantage in this hop.

$$Pr[G_0 = 1] = Pr[G_1 = 1]$$

Game G_2. This game, presented in Figure 6.7, is a two-step hop that alters the process of generating location tokens for every ORAM access. First, every function F is replaced with a real randomly sampled function g. Secondly, the input messages to the functions are replaced by the

Init$(\mathcal{I}, \mathcal{T}, N, L, d)$

$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); g \leftarrow\$\ Func(D, R)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c_S \leftarrow 0; \mathcal{I}_A \leftarrow [][]$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots, d^l\}$ **do**
    $\delta \leftarrow (g(c_S), g(c_S + 1))$
    $d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$
    $\mathcal{I}_A[l][a] \leftarrow d'$
    $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, d')$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o; c_S \leftarrow c_S + 2$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, \ldots, N\}$ **do**
  $\delta \leftarrow (g(c_S), g(c_S + 1))$
  $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \mathcal{T}[i])$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$
  $c_S \leftarrow c_S + 2$
$st \leftarrow (g, \mathsf{sk}_E, c_S, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathcal{I}_A)$
return $(st, (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$

Search$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$

$(g, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S, \mathcal{I}_A) \leftarrow st$
$a \leftarrow 0;$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (g(c_S), g(c_S + 1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
  $data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
  $o_R \leftarrow \mathsf{Next}(\mathcal{I}_A[l][a], \tau)$
  $(data', a', c') \leftarrow o_R$
  $data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data')$
  $o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
  $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W$
  $a \leftarrow a'; c_S \leftarrow c_S + 2$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta \leftarrow (g(c_S), g(c_S + 1))$
$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$data \leftarrow \Theta.\mathsf{Dec}(\mathsf{sk}_E, data_c)$
$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, data)$
$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
$st' \leftarrow (g, \mathsf{sk}_E, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S + 2, \mathcal{I}_A)$
return $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', data)$

InitNode(Node)

$d' \leftarrow []$
for $i \in \{0, 1, \ldots, |Node|\}$ **do**
  $(key, a') \leftarrow Node[i]$
  $d'[i] \leftarrow (key, a', 1)$
return d'

Figure 6.7: Game 2 hop.

current value on the global counters. Since these input messages are unique by construction, we are exchanging unique values by unique values, enabling us to use the global counter in the PRF. For every ORAM access either a tree level l, a node offset a or access counter c is different. As the secret key sk_F is outside of the adversary control, an adversary that can distinguish between G_1 and G_2 can also be used to distinguish F from a truly random function. As such, we upper bound the distance between these two games by building an adversary $\mathcal{B}_1$ against the prf-security experiment such that:

$$Pr[G_1 = 1] - Pr[G_2 = 1] = \mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F},\mathcal{B}_1}(\lambda)$$

Adversary $\mathcal{B}_1$ simulates the game G_2 as follows. For every requested location token the adversary issues a new call to the $\mathsf{prf}_{\mathsf{F},\mathcal{B}_1}$ oracle. Furthermore, the output bit of G_2 is forwarded as the resulting bit of the prf-security experiment. As the difference between both games is the location token generated either by a $\mathsf{F}(l||a||c)$ or $\mathsf{g}(\mathsf{c_S})$, and both input messages are unique, then the probability of distinguishing between game G_2 and G_1 is the same as prf-security.

Game G_3. With game G_3, presented in Figure 6.8, the encrypted data blocks are replaced

Init$(\mathcal{I}, \mathcal{T}, N, L, d)$

$\mathsf{sk}_F \leftarrow F.Gen(1^\lambda); g \leftarrow\$ Func(D, R)$
$\tilde{\mathcal{I}} \leftarrow []; \mathsf{st}_{\tilde{\mathcal{I}}} \leftarrow []; c_S \leftarrow 0; \mathcal{I}_A \leftarrow [][]$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow \Phi.\mathsf{Build}(d^l)$
  for $a \in \{0, 1, \ldots, d^l\}$ **do**
    $\delta \leftarrow (g(c_S), g(c_S + 1))$
    $d' \leftarrow \mathsf{InitNode}(\mathcal{I}[l][a])$
    $\mathcal{I}_A[l][a] \leftarrow d'$
    $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
    $o \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, c_d)$
    $(\mathsf{st}_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}_l) \leftarrow o; c_S \leftarrow c_S + 2$
  $\tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}_l; \mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}_l}$
$(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Build}(N)$
for $a \in \{0, \ldots, N\}$ **do**
  $\delta \leftarrow (g(c_S), g(c_S + 1))$
  $c_d \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
  $(\mathsf{st}_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}) \leftarrow \Phi.\mathsf{Write}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, c_d)$
  $c_S \leftarrow c_S + 2$
$st \leftarrow (g, \mathsf{sk}_E, c_S, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, \mathcal{I}_A)$
return $(st, (\tilde{\mathcal{I}}, \tilde{\mathcal{T}}))$

Search$(st, \tilde{\mathcal{I}}, \tilde{\mathcal{T}}, \tau)$

$(g, \mathsf{sk}_E, \mathsf{st}_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S, \mathcal{I}_A) \leftarrow st$
$a \leftarrow 0;$
for $l \in \{0, 1, \ldots, L\}$ **do**
  $\mathsf{st}_{\tilde{\mathcal{I}}_l} \leftarrow \mathsf{st}_{\tilde{\mathcal{I}}}[l]; \tilde{\mathcal{I}}_l \leftarrow \tilde{\mathcal{I}}[l]$
  $\delta \leftarrow (g(c_S), g(c_S + 1))$
  $(\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \_) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a)$
  $o_R \leftarrow \mathsf{Next}(\mathcal{I}_A[l][a], \tau)$
  $(data', a', c') \leftarrow o_R$
  $data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
  $o_W \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{I}}_l}, \delta), \tilde{\mathcal{I}}_l, a, data'_c)$
  $(\mathsf{st}''_{\tilde{\mathcal{I}}_l}, \tilde{\mathcal{I}}'_l) \leftarrow o_W$
  $a \leftarrow a'; c_S \leftarrow c_S + 2$
  $\mathsf{st}_{\tilde{\mathcal{I}}}[l] \leftarrow \mathsf{st}''_{\tilde{\mathcal{I}}_l}; \tilde{\mathcal{I}}[l] \leftarrow \tilde{\mathcal{I}}'_l$
$\delta \leftarrow (g(c_S), g(c_S + 1))$
$(\mathsf{st}'_{\tilde{\mathcal{T}}}, data_c) \leftarrow \Phi.\mathsf{Read}((\mathsf{st}_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a)$
$data'_c \leftarrow \Theta.\mathsf{Enc}(\mathsf{sk}_E, \{0\}^B)$
$(\mathsf{st}''_{\tilde{\mathcal{T}}}, \tilde{\mathcal{T}}') \leftarrow \Phi.\mathsf{Write}((\mathsf{st}'_{\tilde{\mathcal{T}}}, \delta), \tilde{\mathcal{T}}, a, data'_c)$
$st' \leftarrow (g, \mathsf{sk}_E, \mathsf{st}''_{\tilde{\mathcal{T}}}, \mathsf{st}_{\tilde{\mathcal{I}}}, c_S + 2, \mathcal{I}_A)$
return $(st', \tilde{\mathcal{I}}, \tilde{\mathcal{T}}', \{0\}^B)$

InitNode(Node)

$d' \leftarrow []$
for $i \in \{0, 1, \ldots, |Node|\}$ **do**
  $(key, a') \leftarrow Node[i]$
  $d'[i] \leftarrow (key, a', 1)$
return d'

Figure 6.8: Game 3 hop.

by dummy message with a constant length B. As the contents of the table blocks are never disclosed to the adversary according to the security model and the access counters stored within the tree nodes have been replaced by the global counter, the blocks contents are no longer relevant for the protocol execution. Furthermore, as the adversary $\mathcal{A}$ does not have access to the secret key sk_E the upper bound between game G_4 and game G_3 is defined by an adversary $\mathcal{B}_2$ in an IND-CPA experiment $\mathsf{Adv}^{\mathrm{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda)$.

Adversary $\mathcal{B}_2$ simulates the game G_3 by forwarding for every encryption request a pair of input message $(data, \{0\}^B)$ to the **IND-CPA**$_{\theta,\mathcal{B}_2}$ experiment. After the requests, the protocol continues to be executed with the ciphertext returned from the experiment. Once G_3 terminates its resulting bit is returned as the guessing bit of the experiment. Since the difference between both games is the same as presenting the encryption of one of the input messages, then the probability that $\mathcal{A}$ distinguish between G_2 and G_3 is the same as the IND-CPA security.

$$Pr[G_2 = 1] - Pr[G_3 = 1] = \mathsf{Adv}^{\mathrm{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda)$$

Game G_4**.** In this game, presented in Figure 6.9, we replace every block address a in an

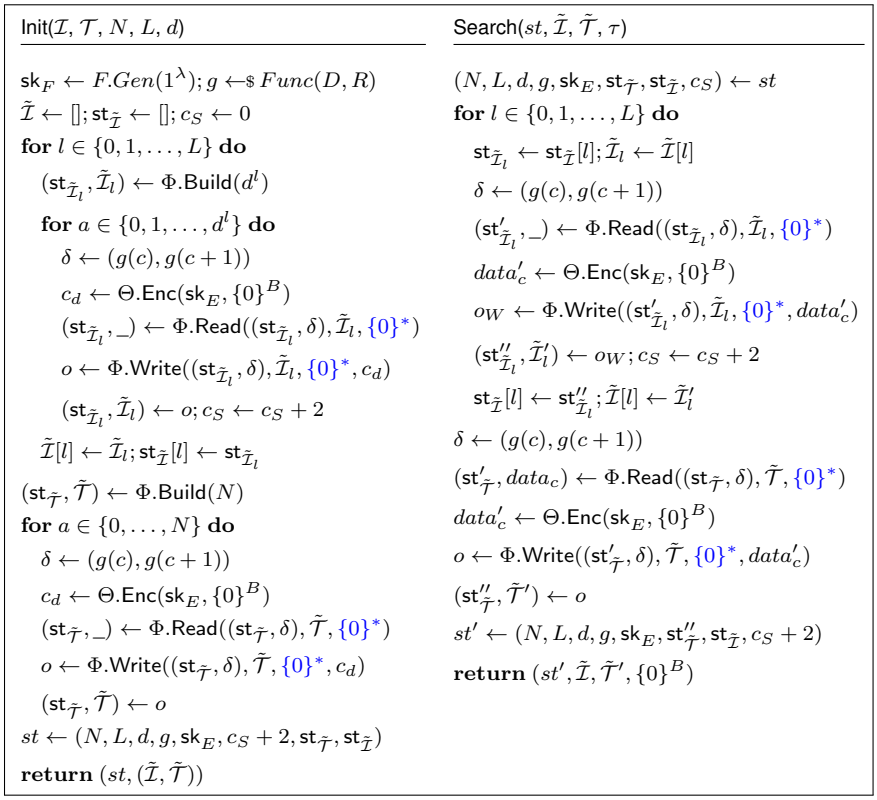


Figure 6.9: Game 4 hop.

ORAM read and write requests by the fixed address 0. With this modification, game G_3 has a data request sequence $\overrightarrow{y_3}$ that depends on the input query and database contents while G_4 has a data request sequence $\overrightarrow{y_4}= ((\cdot, 0, \cdot), \ldots, (\cdot, 0, \cdot))$ that always accesses the same address. Since both requests have the exact same length then according to the security definition of an ORAM construction the access pattern generated by both requests are indistinguishable $\mathcal{X}[\Phi](\overrightarrow{y_3}) \approx \mathcal{X}[\Phi](\overrightarrow{y_4})$. In fact, our constructions leverage an ORAM scheme with an external pmap which determines the access patterns generated. Since in both games the pmap sequence $\overrightarrow{\delta}$ is generated by an oracle $\mathcal{O}$ instantiated as a random function g that outputs a unique message independent of the accessed addressed then the access patterns generated are indistinguishable from an access pattern generated by an ORAM construction with an internal pmap. We upper bound the distance between both games with an hybrid argument [BS17] of an adversary $\mathcal{B}_3$ that plays against an ORAM experiment $\mathsf{Adv}^{\mathrm{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda)$.

In this game we apply a standard hybrid argument where and adversary $\mathcal{B}_3$ has to distinguish between a sequence of $L+1$ hops such that each hop is denoted as $G_{(3,i)}$ where $0 \leq i \leq L+1$. In the first hop $G_{(3,0)}$ everything in the game is identical to G_3, but in the hop $G_{(3,1)}$ the access to the first ORAM level is made to a fixed address 0. As everything remains equal, distinguishing between these two hops is the same as distinguishing between two access patterns generated

by an ORAM construction. This argument can be applied recursively in the subsequent levels by changing one level at each step, from $G_{(3,i)}$ to $G_{(3,(i+1))}$ where $i \in [0..L]$. In the last game, $G_{(3,(L+1))}$ is exactly equal to G_4. As such, the advantage of an adversary $\mathcal{A}$ distinguishing between G_4 and G_3 is the same as $\mathcal{B}_3$ distinguishing the access patterns generated of just one of the $L+1$ ORAM levels in the sequence of from $G_{(3,0)}$ to $G_{(3,(L+1))}$. The upper bound is given by:

$$Pr[G_3 = 1] - Pr[G_4 = 1] = (L+1) \cdot \mathsf{Adv}^{\mathrm{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda)$$

Let,

$$\mathsf{Adv}^{\mathrm{OBLIVS}}_{OIS,S,\mathcal{A}}(\lambda) = \left| \Pr\left[\mathbf{Real}^{OIS}_{\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}^{OIS}_{S,\mathcal{A}}(1^\lambda) = \right] \right|$$

then Theorem 6.6.1 follows from

$$\mathsf{Adv}^{\mathrm{OBLIVS}}_{OIS,S,\mathcal{A}}(\lambda) = \sum_{i=0}^{4} |Pr[G_i = 1] - Pr[G_{i+1} = 1]| \leq$$
$$\mathsf{Adv}^{\mathrm{prf}}_{\mathsf{F},\mathcal{B}_1}(\lambda) + \mathsf{Adv}^{\mathrm{IND-CPA}}_{\Theta,\mathcal{B}_2}(\lambda) + L + 1 \cdot \mathsf{Adv}^{\mathrm{ORAM}}_{\Phi,\mathcal{B}_3}(\lambda) \leq \mathsf{negl}(\lambda)$$

■

6.7 Evaluation

We implemented CODBS as a PostgreSQL server-side extension that supports equality and range queries. We build upon on a widely used open-source database management systems to ensure that our system design choices are based on realistic assumptions and the evaluation results are comparable to industry standard databases. Furthermore, this approach enables us to provide a turnkey solution that can be easily integrated with existing applications. The complete solution has roughly 12K lines of C code and is composed by an ORAM library, a *Trusted Proxy* engine and a database wrapper.

6.7.1 System Implementation

Our system currently supports two ORAM constructions: Path ORAM and Forest ORAM. We implemented both constructions in a general-purpose ORAM library, open-source for any application that needs to hide its access patterns. The library has no third-party dependencies (e.g.: Intel SGX, OpenSSL) and decouples the main logic of an ORAM algorithm from its auxiliary data structures (stash, position map and external storage). With this design, an application can

include our library as a static or shared library and customize the low-level accesses to the external storage.

We implemented CODBS as the database component that replaces the *Trusted Proxy* and provides an input API similar to the definition in Section 6.3.1. Additionally, this component has an output API to access the external database storage. The component is deployed within an Intel SGX enclave collocated with the database. Currently, the extension supports a B^+-tree as the index data structure. We leverage the LibSodium [Den20] library v1.0.5 to instantiate the cryptographic primitives as it provides constant-time implementations. Concretely, we instantiate the PRF F as a SHA256-HMAC and Θ encryption scheme as an AES block cipher with CBC mode.

The *Trusted Proxy* is connected to the database with a wrapper component implemented as Foreign Data Wrapper (FDW), a PostgreSQL module that enables developers to extend the database server without modifying the core source code. With this component, client applications can define oblivious tables that are not accessed by the standard PostgreSQL execution path but instead by user-defined functions. In our system, the wrapper intercepts input queries and forwards them to the *Trusted Proxy* trough ECALLs to the enclave. For each query, the *Trusted Proxy* generates a sequence of oblivious invocations to the database storage using OCALLs. These invocations are handled by the database wrapper which accesses the database physical storage using the PostgreSQL low-level table interface. An additional advantage of an FDW module is the transparent integration of CODBS with a database client. A database client can establish an oblivious stream of requests with the *Trusted Proxy* trough database cursors [Mac16]. A cursor is a query control data structure that enables the client to fetch a few rows at a time from a specific query instead of obtaining every resulting row. Using this mechanism, the client can keep pooling query results from the *Trusted Proxy* and issuing new queries.

6.7.2 Methodology

We measure the performance of our system to answer the following questions: *1)* How does CODBS scale with increasingly larger datasets; *2)* What is the overhead in comparison to a plaintext database for different types of queries; *3)* How does size of the result set of a range query impact the overall system the database performance. In the evaluation we compare our construction to a system `Baseline` which consists of a database that stores the *Table Index* and *Table Heap* in a single Path ORAM construction. For a fair comparison, the `Baseline` also uses an oblivious query stream.

Micro & Macro Settings. We divided our system evaluation in two distinct settings, a micro setting and a macro setting. Both settings use a synthetic dataset and workload. The micro

setting measures the performance of Forest ORAM construction and Path ORAM constructions isolated from the CODBS scheme and the PostgreSQL engine. In this setting each construction read/writes blocks of $B = 8$ KiB from/to the main memory at randomly sampled positions. The data blocks written to memory are also sampled from a uniform distribution $\{0, 1\}^B$. In the macro setting we use the YCSB benchmark v0.18 [CST+10]. In the benchmark the database has a single table with two columns. The first column Key is indexed and stores unique keywords. The second column stores JSON objects containing randomly sampled data. Each table record has the same size as a database block 8 KiB. We configured the benchmark to generate two workloads over the indexed column: **Workload A)** Equality queries that search for keywords sampled from a uniform random distribution; **Workload B)** Range queries that start on a randomly sampled keyword and search for at most k values where k is uniformly sampled from $[1..X]$. The first benchmark is designed with a one-to-one match between a database record and database table block to enable a linear analysis of the expected database performance as the table size increases.

For both benchmarks we performed 5 runs for each combination of deployment, configuration, workload and database size. The number of runs is the maximum necessary to calculate an accurate Confidence Interval (CI) [Jai91] with the measured standard deviation. Each run lasted for 40 minutes with a 10-minute warm up period and a 2-minute cool down period between each run. Furthermore, we ensure that each run is an independent observation by clearing the systems caches and deleting any persistent data.

Collected Metrics. In the micro benchmark we measure the mean and the percentile latencies of a read and write operations for every run. With the YCSB benchmark we collect the mean and percentile latencies as well as the system throughput for every run. The samples mean are calculated within an 95% CI with the Student's t-distribution [Jai91]. We collected CPU, memory and disk usage of each system and ORAM construction using Dstat v0.7.3.

Experimental Setup. The system was deployed in a private infrastructure. Each computational node had an Intel Core i3-7100 CPU with a clock rate of 3.90 GHz and 2 physical cores in hyper-threading. The main memory was a 16 GiB DDR3 RAM and the solid-state storage a Samsung PM981 NVME with 250 GB. The machines had Intel SGX SDK v2.0 installed. Additionally, the nodes were interconnected by a 10 GiB network switch.

6.7.3 Micro-benchmark

Figure 6.10 depicts the results of the micro-benchmark. The workload in this benchmark with an initially empty oblivious data structure and measures the latency of an oblivious access request, either a read or a write. With this workload we measure the average latency of a request for the

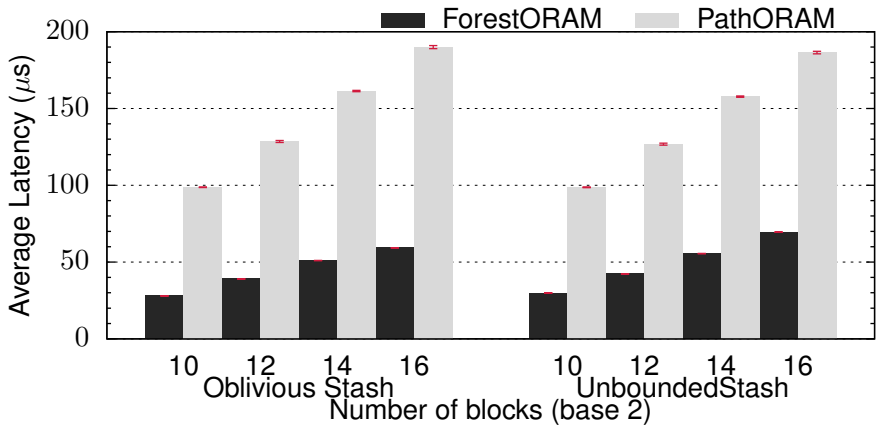


Figure 6.10: Forest ORAM and Path ORAM comparison. X-axis measures number of blocks and errors bars the 95% CI.

Forest ORAM and Path ORAM constructions. We also measure the latency of both constructions with two distinct stashes, an unbounded stash where a stash access stops as soon as it finds an element and a double-oblivious stash with fixed size upper bounded at $\log(N)$. The number of blocks stored on the ORAMs increases from 2^{10}(65 MiB) to 2^{16} (2 GiB).

As can be observed, the performance difference between both stashes is almost non-existent. This is expected as position-based ORAM constructions are designed to utilize as much as possible the stash. In more detail, on an oblivious stash a Forest ORAM request takes on average 27 μs for the smallest data set while in the largest takes 59 μs. The Path ORAM has a higher latency, with 99 μs for an oblivious request in the smallest data set and 190 μs for the largest data set. This difference represents at least a $\sim 2.6\times$ speedup. In the unbounded stash, the most significant difference is noticed on Forest ORAM in the 2^{14} dataset where there is a an average performance decrease of 2.5%. As the dataset increases, the performance of both systems degrades at a similar rate with Forest ORAM latency increasing by $\sim 15\%$ and Path ORAM by $\sim 17\%$. At the 90th percentile, both systems performances degrade considerably, with Forest ORAM latency increasing at most by $\sim 17\%$ and Path ORAM by 7%. Even with these outliers Forest ORAM latency is at least $\sim 1.6\times$ lower than Path ORAM.

This benchmark shows that the asymptotic difference between Forest ORAM and Path ORAM has a practical impact. The average latency as well as the 90th and 99.9th percentiles are smaller than Path ORAM. This difference is attributed to the partition framework which scales the number of partitions and the tree height of each individual partition as the data set increases. In fact, the number of partitions depends on parameter that can be adjusted to increase even further the performance of Forest ORAM at the cost of additional client-side storage. The only unexpected result is the 99.9th percentile maximum performance degradation of $\sim 120\%$ when compared to Path ORAM. However, this difference only occurs in the smallest data sizes and

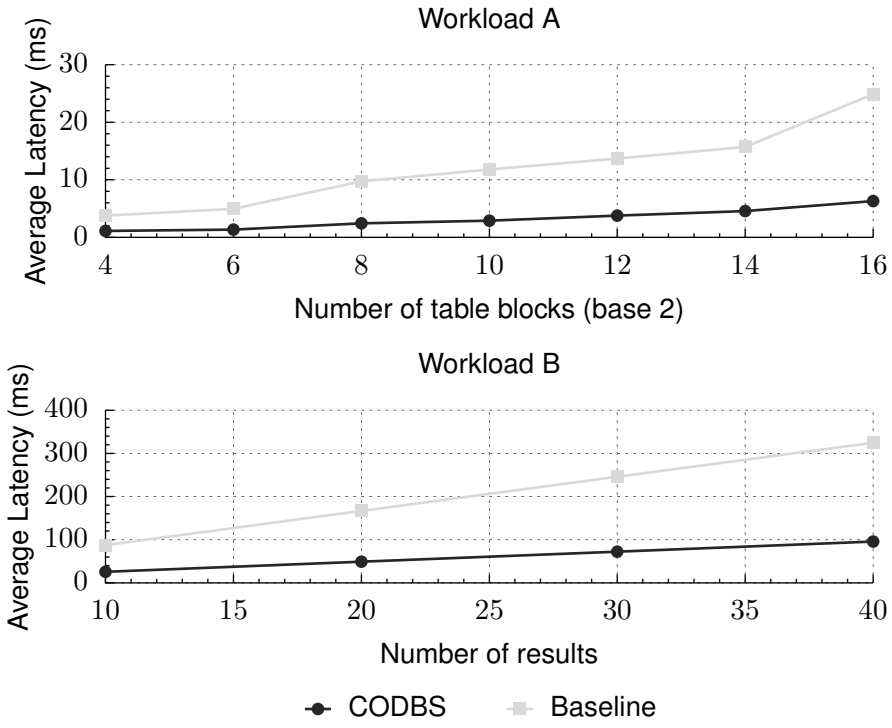


Figure 6.11: Avg. latency of YCSB workloads. Workload A X-axis measures the numbers of blocks. Workload B X-axis measures a query resulting records.

stabilizes in both protocols at $\sim 40\%$.

6.7.4 Macro-benchmark

We now present the performance of a complete CODBS deployment and compare it to `Baseline`. The `Baseline` solution uses a Path ORAM construction to store the database data and does not divide the *Table Index* in multiple ORAM levels. Instead, the *Table Index* is stored in a single ORAM and accessed as an oblivious data structure similar to the one used in Oblix and proposed by *Wang et. al* [MPC+18, WNL+14]. However, it still calculates the block addresses using a PRF to keep both systems comparable. With this approach, the `Baseline` provides clear understanding on the practical performance improvements of our cascade solution. Additionally, we also contrast both solutions to a plaintext PostgreSQL database. The evaluation consists on measuring the throughput and latency of increasingly larger database databases until a saturation point is reached and the systems cannot provide a practical throughput ($>$ 1 op/s). Besides the performance of database searches, we also measure the cost of outsourcing the initialization algorithm.

Figure 6.11 presents the macro results. In workload A, the database size starts with 2^4 *Table Heap* records and a *Table Index* with a single tree level (a total of 47 MiB) and is increased until 2^{16} *Table Heap* blocks with a *Table Index* of two levels (a total 2.1 GiB). Across this range,

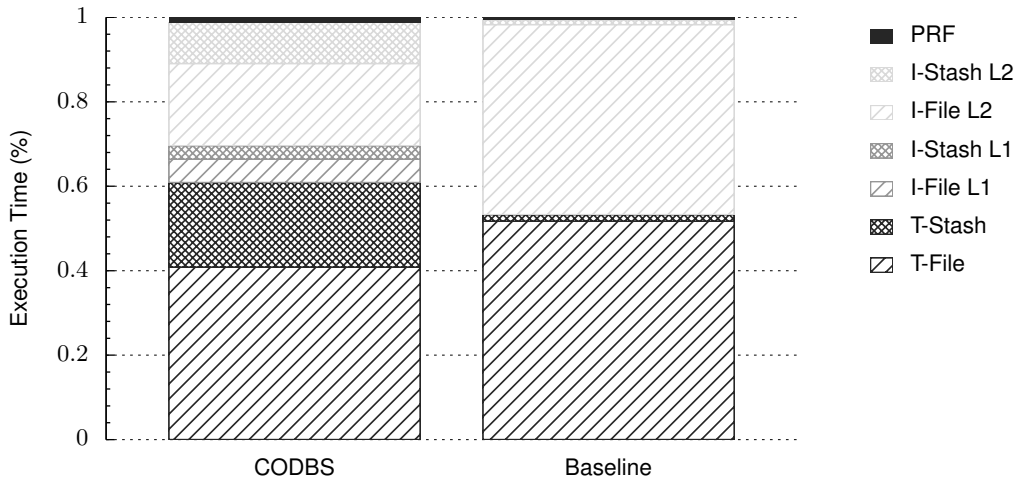


Figure 6.12: Breakdown of time spent during query execution.

CODBS maintains an average latency below 10 ms which corresponds to 886 ops/s for the smallest data set and 158 ops/s for the largest. The average maximum throughput of every run in `Baseline` is 264 ops/s (smallest dataset) and the average latency surpasses CODBS at just 2^8 *Table Heap* records. Its highest average latency is 25 ms, corresponding to a throughput of 40 ops/s, meaning that CODBS has approximately a $4\times$ speedup. There is a slight performance degradation of both systems on the 99th percentile in the largest dataset. CODBS has a 30% latency increase with an average latency of 13.34 $\pm$ 1.74 ms and the `Baseline` has an increase of 17% with an average latency of 29.33 $\pm$ 12.95 ms. In contrast to both solutions, a plaintext PostgreSQL has on average ~ 7663 ops/s.

Workload B uses the dataset with 2^{16} *Table Heap* records to measure the latency of range scans, more specifically a where clause with a greater than operator. The number of returned records ranges from a 10 to 40, with larger ranges becoming impractical. Similar to workload A, the `Baseline` system has the highest average latency of 324 ms and a throughput of 3 ops/s. CODBS has a $\sim 3\times$ speedup with the lowest throughput of 10 ops/s and an average latency of 96 ms. While both solutions have a considerable performance decrease, a plaintext PostgreSQL has at most a decrease of $\sim 26\%$ on the largest dataset from returning a single result to return 40 results.

Figure 6.12 provides an analysis of the multiple query processing stages in both systems. It breakdowns the execution between the database data structures, *Table Index* (I-File, I-Stash) and the *Table Heap* (T-File, T-Stash) and PRF computation. Each structure is divided even further by the time spent in the ORAM stash and external access to store (I-File, T-File). The CODBS breakdown also accounts for the time spend at each index level (L2 and L1). The overhead of block encryption is measured within the accesses to the external files. As depicted, CODBS spends most of the time (60%) accessing the *Table Heap*, 8% accessing the first *Table Index* level and 28% accessing the second *Table Index* level and the remaining time calculating the

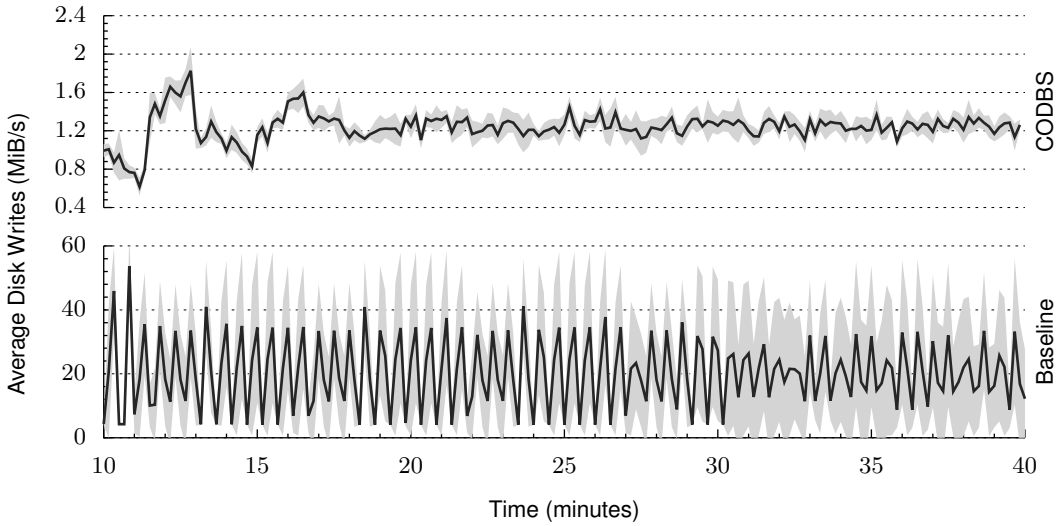


Figure 6.13: Avg. disk writes over time on workload A on data set with 2^{16} records. The light gray represents the 95% CI.

PRFs. In contrast, the `Baseline` accesses are almost equally divided between the *Table Heap* and *Table Index*. With the `Baseline` spending more time accessing the external file, the system throughput is dominated by the disk IO. This claim is further supported by Figure 6.13 which presents the average write requests to the external storage grouped in intervals of 10 seconds. As can be observed, the `Baseline` has a sustained rate 10 to 40 MiB writes per second while CODBS is constantly below 2 MiB/s.

We also measure the performance and storage penalties associated with the precomputation of the initialization algorithm. For the dataset with 2^4 records CODBS has an average latency of 23 seconds to initialize 47 MiB. The `Baseline` takes on average 28 seconds to the complete the initialization. On the largest dataset, CODBS initialization latency is $\sim 14\times$ higher with an average of 346 seconds while `Baseline` is $\sim 18\times$ higher. Both solutions incur in storage overhead with a negligible growth on the smallest dataset but with a significant growth in largest datasets. For a plaintext database with 2^{16} records (2.1 GiB), `Baseline` has a threefold increase (6.2 GiB) while CODBS is $\sim 5\times$ larger (11 GiB).

Discussion. Across every benchmark and workload, CODBS displays an overall performance that exceeds that of the `baseline` system. These speedups are the result of combining the cascade approach with the Forest ORAM construction. This combination results in an asymptotic decrease of $\log(\log(N))$ bandwidth blowup compared to state-of-the-art oblivious data structures as shown in Section 6.7.3. This seemingly small difference has a significant impact.

In the YCSB benchmark on workload A with a tree height of just two levels there is a $4\times$ speedup instead of just a $2.6\times$ speedup as might otherwise be expected from the micro benchmarks. This difference is the result of spending less time accessing the storage and a

95% lower number of disk writes on average than the `baseline`. Regarding workload B the performance gains of CODBS in comparison to the `baseline` are less significant. With just a $2\times$ speedup, the main bottleneck in this workload seems to be the size of the data exchanged in the oblivious query stream. Across the different result set size, CODBS has on average a write rate of ~ 1.3 MiB/s and the `baseline` writes at most ~ 347 KiB/s.

6.8 Summary

Along this chapter we presented a novel cryptographic scheme for database query searches that minimizes the information leaked to a few public parameters. The common approach in the state-of-the-art is to protect sensitive data on databases by using cryptographic schemes as a black-box. However, this approach results in disclosing partial information when the schemes are applied without taking into consideration the data structures of relational systems. In this chapter, we take the opposite approach and propose a novel construction that leverages the query-optimized data structures of databases. We present CODBS, an efficient solution for secure database searches on tree-based indexes and heap table accesses. Our main contribution is a cascade of multiple ORAM levels, each storing blocks from a database index or table. We also use a general construction that removes the need for an ORAM position map and instead use an external map that can be computed on demand during a query search. The overhead of database search is reduced to the bandwidth of accessing multiple small ORAMs instead of accessing a large ORAM multiple times. We further improve the performance of database searches by resorting to an IEE (Intel SGX) collocated with the database engine. Our construction hides the internal memory access patterns of trusted hardware and we prove its security in the style of provable security.

Our solution shows that taking into consideration application specific details can result in more efficient cryptographic schemes. We implemented our construction as a system integrated with a PostgreSQL database and measured its performance. Comparatively to the state-of-the-art constructions, our solution is $1.2\times$ to $4\times$ faster and only requires a small constant size storage that can be deployed within an Intel SGX enclave. A straightforward line of research is the practical application and improvement of our solution on applications that are read-intensive and can fully leverage our cascade solution. Conceptually, it's possible to have a parallel system that accesses multiple ORAM concurrently and increases the system throughput. Another alternative is to enhance our construction with dynamic updates and do a security analysis of the information that is disclosed.

Chapter 7

Conclusion

The Cloud infrastructure is the *de facto* standard technology for migrating on-premises applications to a remote third-party service. The main benefit of cloud providers is the seemingly virtual infinite resources that can be allocated on-demand at a lower cost than classical data-centers infrastructures. Having a cost-effective third-party service that manages the hardware as well as software enables individuals and organizations alike to focus on their products. This has resulted on several sectors offloading their applications to clouds such as e-commerce portals, social network, streaming services and retail chains. However, organizations that handle sensitive data such as medical applications or financial services refrain from fully adopting the cloud paradigm due to security concerns. The main detracting factor is the immediate loss of data control and possible data disclosures.

CPD have been proposed to protect the user's confidentiality. These systems encrypt data before offloading it an untrusted third-party service. However, classical encryption schemes limit the database functionality which is essential to several different applications. To mitigate this issue, existing systems prioritize performance and functionality over privacy and leak partial information such as the order or equality between values. This leakage enables the database engine to process queries over ciphertexts on the server-side with minimal overhead or functional limitations. In this dissertation we proposed, developed and evaluated three new novel solutions to store and process confidential data in untrusted services. Our aim was to explore alternative trade-offs between privacy, performance and functionality in CPD systems. This goal was achieved with our contributions that offload increasingly more computation to the untrusted server. All of our contributions consider a semi-honest, static adversary that observes everything that is stored and evaluated on the untrusted service.

Our first contribution provides a multi-layer, stackable user space encrypted file system to outsource sensitive data. In this contribution, the database engine remains in a private infrastructure, but sensitive data is permanently stored on an untrusted site. Our solution, SafeFS, intercepts database request to the file system and modifies them in a pipeline of layers

that can encrypt, replicate and compress data. To support these features, the system was designed with the concept of modular layers that encapsulate a single feature. Layers can be composed on top of each other by following a standard input and output interface. The proposed system was evaluated experimentally, and the results show that SafeFS is a practical alternative to tailor-fit solutions.

Even tough cloud providers have safeguards to prevent an external attacker from compromising a system, there are always exploits. Our second contribution leverages the security of existing providers and proposed a novel system, d'Artagnan, that decentralizes private information. Our system uses secret sharing to encrypt data in multiple secrets that can be stored in non-colluding third-parties. With this approach, the corruption of a single party does not compromise the security of the database and the original values can only be reconstructed if the majority of the parties are corrupted. As such, an adversary has to commit far more resources on an attack than in a classical database. d'Artagnan also manages multiple computing nodes to ensure a consistent global state and evaluate queries using secure multiparty protocols. The system has a modular architecture that is agnostic to the underlying protocols and can integrate new constructions to support active adversaries, a dynamic set of participating parties and provide higher availability. The system was validated with an experimental evaluation that show that d'Artagnan is a viable solution for small, but highly sensitive datasets.

Our third core contribution took a more in-depth approach to hide the access-patterns disclosed by queries in relational databases. We proposed a new oblivious index scan scheme that reduces the information disclosed by an index scan to a few public parameters such as the height and fanout of a search tree. Furthermore, we presented a novel constant-time oblivious construction that can be securely deployed on trusted-hardware and has a bandwidth blowup lower than state-of-the-art constructions. Our solution uses an optimized ORAM scheme, Forest ORAM, that improves the practical performance and scalability of tree-based ORAM constructions. We integrated the novel constructions with one of the most widely used open-source database systems, PostgreSQL, and measured the resulting system throughput. The obtained results show that our approach is at least twice as fast as existing solutions.

Overall, the contributions in this dissertation are practical instances of systems that answer positively to our research questions. The proposed systems support an extensive set of queries without an underlying assumption that partial information needs to be disclosed. However, each system has some constraint either due to the trust model or the cryptographic schemes used. SafeFS provides a practical and efficient solution to outsource data but still discloses the underlying access patterns of the database and assumes the client has the computational resources to host a database engine in a private infrastructure. The next contribution, d'Artagnan, can be used to offload any computation to an untrusted third-party but the existing secure multiparty protocols have a significant overhead that is not practical for large-scale systems. One important insight from the cloud evaluation is the impact that the location of the clouds has on

the system throughput. Ideally, the infrastructure of the cloud providers should be geographically close and have at least a 10 GiB network between the sites. Nonetheless, even if the bandwidth usage of the protocols is reduced to an optimal number of messages, this decentralized approach will always be slower than any centralized solution. However, this may be an acceptable cost depending on an application performance requirements. Our last contribution, CODBS, found a balance between performance and functionality without compromising the user's confidentiality. This solution makes the explicit assumption that there is a trusted proxy (IEE) on a cloud provider that cannot be compromised. However, this assumption may not always be valid as not all cloud providers have computing instances with trusted hardware. In these cases, one of our previous contributions might be preferable. A general purpose CPD with practical performance that is not susceptible to statistical or side-channel attacks is currently unfeasible and there are still several research directions left open.

Future Work. An immediate accessible research path is improving our contributions by applying them to practical uses cases and proposing or integrating new cryptographic schemes that provide interesting trade-offs between performance and confidentiality. An example is the integration of ORAM schemes on SafeFS to hide the database access patterns. From a system design perspective, it is not entire clear how these schemes can be integrated in a stackable architecture. Furthermore, the system performance will decrease significantly and it will require new solutions to mitigate the overhead. A possible research path is to propose novel optimized ORAM constructions that follow an approach similar to Forest ORAM and distribute data between multiple, smaller ORAMs.

In a different direction there is an immediate need for an empirical study on access pattern leakage of specific applications. SSE schemes as well as CODBS have taken a best-effort approach to minimize the information disclosed by search queries. To further improve the existing lower bounds, it is critical to have an in-depth knowledge of what is an acceptable leakage in different application settings. Towards this goal there is the need for representative datasets and workloads of application in specific domains that manage sensitive information.

Even though we presented a straightforward solution to the problem of volume leakage in Chapter 6, it is still a challenging open problems without an efficient solution. The current approaches to address this issues resort to changing the distribution of values in a database by adding dummy elements, limit searches to a fixed size or using differential privacy. In our last contribution we opted for maintaining a constant stream of requests between the clients and the database engine to prevent an adversary from discerning real requests from dummy requests. Exploring our approach in applications that have queries running periodically may provide some insights on how to minimize the number of dummy requests and increase the system throughput. Solving this problem with an efficient solution is one of the current main limitations to create a general-purpose CPD.

Chapter 8

Bibliography

[AAB+14] A. Alba, G. Alatorre, C. Bolik, A. Corrao, T. Clark, S. Gopisetty, R. Haas, R. I. Kat, B. S. Langston, N. S. Mandagere, D. Noll, S. Padbidri, R. Routray, Y. Song, C.-H. Tan, and A. Traeger. Efficient and agile storage management in software defined environments. *IBM J. Res. Dev.*, 58(2–3):5, March 2014. (**Cited** on page 54.)

[ABB+16] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, August 2016. USENIX Association. (**Cited** on pages 103, 116 and 121.)

[ABF+17] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries — Breaking the 1 Billion-Gate Per Second Barrier. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 843–862, May 2017. (**Cited** on page 95.)

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. (**Cited** on page 12.)

[AKL+18] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, and Elaine Shi. OptORAMa: Optimal Oblivious RAM. *IACR Cryptology ePrint Archive*, 2018:892, 2018. (**Cited** on page 107.)

[Ama] Amazon. AWS. (**Cited** on pages 53 and 89.)

[BCQ+13] Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *TOS*, 9:12:1–12:33, 2013. (**Cited** on page 43.)

[BGC+18] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The Tao of Inference in Privacy-Protected Databases. *PVLDB*, 11(11):1715–1728, 2018. (**Cited** on page 78.)

[BGG+09] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, 2009. Association for Computing Machinery. (**Cited** on page 54.)

[BHJP14] Christoph Bösch, Pieter H. Hartel, Willem Jonker, and Andreas Peter. A Survey of Provably Secure Searchable Encryption. *ACM Comput. Surv.*, 47:18:1–18:51, 2014. (**Cited** on page 14.)

[BKN04] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Trans. Inf. Syst. Secur.*, 7(2):206–241, 2004. (**Cited** on page 25.)

[Bla93] Matt Blaze. A cryptographic file system for unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, page 9–16, New York, NY, USA, 1993. Association for Computing Machinery. (**Cited** on page 43.)

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag. (**Cited** on pages 49 and 88.)

[BMO+14] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A Shared Cloud-backed File System. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, Philadelphia, PA, June 2014. USENIX Association. (**Cited** on pages 43 and 54.)

[BMO17] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1465–1482, New York, NY, USA, 2017. Association for Computing Machinery. (**Cited** on pages 27 and 45.)

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*,

page 991–1008, Baltimore, MD, August 2018. USENIX Association. (**Cited** on page 49.)

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012. (**Cited** on page 88.)

[Bog13] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, 2013. (**Cited** on page 33.)

[Bol09] Boldyreva, Alexandra and Chenette, Nathan and Lee, Younho and O'Neill, Adam. Order-Preserving Symmetric Encryption. In Joux, Antoine, editor, *Advances in Cryptology - EUROCRYPT 2009*, pages 224–241, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. (**Cited** on page 26.)

[BPF+16] Dorian Burihabwa, Rogerio Pontes, Pascal Felber, Francisco Maia, Hugues Mercier, Rui Oliveira, João Paulo, and Valerio Schiavoni. On the cost of safe storage for public clouds: An experimental evaluation. In *35th IEEE Symposium on Reliable Distributed Systems, SRDS 2016, Budapest, Hungary, September 26-29, 2016*, pages 157–166. IEEE Computer Society, 2016. (**Cited** on page 18.)

[BPSW16] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 245–260, 2016. (**Cited** on pages 34 and 122.)

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 207, 2005. (**Cited** on pages 63, 104 and 122.)

[Bri18] Miles Brignall. Amazon hit with major data breach days before Black Friday. https://www.theguardian.com/technology/2018/nov/21/amazon-hit-with-major-data-breach-days-before-black-friday, 2018. Online; accessed 30-May-2020. (**Cited** on page 13.)

[BS11] Sumeet Bajaj and Radu Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 205–216. ACM, 2011. (**Cited** on page 49.)

[BS17] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. page 818, 2017. (**Cited** on pages 122 and 128.)

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre:

Exploiting Speculative Execution through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery. (**Cited** on page 49.)

[BWK+17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, August 2017. USENIX Association. (**Cited** on page 103.)

[Cat11] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.*, 39(4):12–27, May 2011. (**Cited** on pages 79 and 80.)

[CCP+20] Hugo Carvalho, Daniel Cruz, Rogério Pontes, João Paulo, and Rui Oliveira. On the Trade-Offs of Combining Multiple Secure Processing Primitives for Data Analytics. In Anne Remke and Valerio Schiavoni, editors, *Distributed Applications and Interoperable Systems*, pages 3–20, Cham, 2020. Springer International Publishing. (**Cited** on page 19.)

[CCSP01] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Giuseppe Persiano. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *USENIX Annual Technical Conference, FREENIX Track*, 2001. (**Cited** on page 43.)

[CD05] Ronald Cramer and Ivan Damgård. *Multiparty Computation, an Introduction*, pages 41–87. Birkhäuser Basel, Basel, 2005. (**Cited** on page 32.)

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008. (**Cited** on page 80.)

[CGJ+09] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, page 85–90, New York, NY, USA, 2009. Association for Computing Machinery. (**Cited** on page 12.)

[CGKO11] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *J. Comput. Secur.*, 19(5):895–934, 2011. (**Cited** on page 46.)

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, NY, USA, 2015. ACM. (**Cited** on page 98.)

[CHBG10] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and Krishna P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *in ICWSM '10: Proceedings of international AAAI Conference on Weblogs and Social*, 2010. (**Cited** on page 73.)

[CJJ+14] David Cash, Joseph Jaeger, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, and Michael Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *In Network and Distributed System Security Symposium (NDSS '14*, 2014. (**Cited** on page 46.)

[CL17] Ran Cohen and Yehuda Lindell. Fairness Versus Guaranteed Output Delivery in Secure Multiparty Computation. *J. Cryptology*, 30(4):1157–1186, 2017. (**Cited** on page 87.)

[CM05] Yan-Cheng Chang and Michael Mitzenmacher. Privacy Preserving Keyword Searches on Remote Encrypted Data. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 442–455, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. (**Cited** on page 45.)

[cry] CryFS. https://www.cryfs.org/ (**Cited** on pages 43 and 65.)

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2010. (**Cited** on pages 73, 90, 101 and 131.)

[DDC16] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What else is revealed by order-revealing encryption? In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1155–1166. ACM, 2016. (**Cited** on page 77.)

[Den20] Frank Denis. The Sodium cryptography library, Fev 2020. (**Cited** on page 130.)

[Dig] Digital Ocean. Digital Ocean. (**Cited** on page 89.)

[DLP+20] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious Coopetitive Analytics Using Hardware Enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. (**Cited** on page 50.)

[DPCCM13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013. (**Cited** on page 68.)

[DWP00] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceeding 2000 IEEE Symposium on Security and Privacy*, pages 44–55, 2000. (**Cited** on page 45.)

[EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.*, 2(2-3):70–246, 2018. (**Cited** on page 32.)

[enc] EncFS. `https://github.com/vgough/encfs`. (**Cited** on pages 43, 54, 61 and 65.)

[EUd18] REGULATION (EU) 2016/679 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 27 April 2016, 2018. (**Cited** on page 13.)

[EZ19] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.*, 13(2):169–183, 2019. (**Cited** on page 99.)

[FBB+18] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. Hardidx: Practical and secure index with SGX in a malicious environment. *J. Comput. Secur.*, 26(5):677–706, 2018. (**Cited** on page 102.)

[FPO+19] B. Ferreira, B. Portela, T. Oliveira, G. Borges, H. Domingos, and J. Leitão. BISEN: Efficient Boolean Searchable Symmetric Encryption with Verifiability and Minimal Leakage. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 103–10309, 2019. (**Cited** on page 46.)

[FVY+17] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham. SoK: Cryptographically Protected Database Search. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191, 2017. (**Cited** on pages 13 and 14.)

[FW19] Emily Flitter and Karen Weise. https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html. https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html, 2019. Online; accessed 30-May-2020. (**Cited** on page 13.)

[Gar19] Sandra E. Garcia. Data Breach at Wyze Labs Exposes Information of 2.4 Million Customers. https://www.nytimes.com/2019/12/30/business/wyze-security-camera-breach.html, 2019. Online; accessed 30-May-2020. (**Cited** on page 13.)

[Gen09] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, 2009. (**Cited** on page 30.)

[GLMP18] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump Up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 315–331, New York, NY, USA, 2018. ACM. (**Cited** on pages 45 and 103.)

[GLMP19] P. Grubbs, M. Lacharité, B. Minaud, and K. G. Paterson. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1067–1083, 2019. (**Cited** on page 45.)

[GMP16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *CRYPTO*, 2016. (**Cited** on page 100.)

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, 1987. (**Cited** on page 64.)

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996. (**Cited** on page 99.)

[Goh03] Eu-Jin Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003. (**Cited** on page 45.)

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004. (**Cited** on pages 16, 32 and 40.)

[Goo] Google. Google cloud plaform. (**Cited** on pages 53 and 89.)

[Gre14] Glenn Greenwald. *No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State*. Metropolitan Books, New York, NY, USA, 2014. (**Cited** on page 13.)

[Hal10] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. 2010. (**Cited** on pages 42, 54 and 65.)

[Hau19] Christine Hauser. EasyJet Says Cyberattack Stole Data of 9 Million Customers. https://www.nytimes.com/2020/05/19/business/easyjet-hacked.html, 2019. Online; accessed 30-May-2020. (**Cited** on page 13.)

[hba] HBase. https://hbase.apache.org. (**Cited** on page 87.)

[HOJY19] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *PoPETs*, 2019(1):172–191, 2019. (**Cited** on pages 99, 102, 114, 116 and 121.)

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012. (**Cited** on pages 16 and 98.)

[Jai91] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991. (**Cited** on page 131.)

[KM17] Seny Kamara and Tarik Moataz. Boolean Searchable Symmetric Encryption with Worst-Case Sub-linear Complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 94–124, Cham, 2017. Springer International Publishing. (**Cited** on pages 41 and 46.)

[KPR12] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 965–976, New York, NY, USA, 2012. Association for Computing Machinery. (**Cited** on page 46.)

[KPT19] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1033–1050, 2019. (**Cited** on page 44.)

[KS14a] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 506–525, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. (**Cited** on page 87.)

[KS14b] Florian Kerschbaum and Axel Schroepfer. Optimal Average-Complexity Ideal-Security Order-Preserving Encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 275–286, 2014. (**Cited** on page 44.)

[KTV+18] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: policy enhanced secure object store. In *EuroSys*, pages 25:1–25:17. ACM, 2018. (**Cited** on pages 90 and 94.)

[les] LessFS. http://www.lessfs.com/wordpress/. (**Cited** on page 43.)

[LMP18] M. Lacharité, B. Minaud, and K. G. Paterson. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314, 2018. (**Cited** on pages 44 and 103.)

[LY81] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981. (**Cited** on page 102.)

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery. (**Cited** on page 34.)

[Mac16] Chris A Mack. *PostgreSQL Development Essentials*. Manpreet Kaur, Baji Shaik, 2016. (**Cited** on pages 67 and 130.)

[Mel98] Jim Melton. *Database Language SQL*, pages 103–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. (**Cited** on page 79.)

[Mica] Microsoft. Always Encrypted (Database Engine). (**Cited** on page 77.)

[Micb] Microsoft. Microsoft Azure. (**Cited** on pages 53 and 89.)

[Mik05] Miklos Szeredi. Filesystem in Userspace, 2005. (**Cited** on page 42.)

[MPC+18] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An Efficient Oblivious Search Index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, May 2018. (**Cited** on pages 99, 101, 103, 116, 121 and 133.)

[MPP+17] Ricardo Macedo, João Paulo, Rogerio Pontes, Bernardo Portela, Tiago Oliveira, Miguel Matos, and Rui Oliveira. A Practical Framework for Privacy-Preserving NoSQL Databases. In *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*, pages 11–20. IEEE Computer Society, 2017. (**Cited** on page 18.)

[MR92] Silvio Micali and Phillip Rogaway. Secure Computation (Abstract). In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, 1992. (**Cited** on page 48.)

[mys] MySQL Enterprise Transparent Data Encryption (TDE). https://www.mysql.com/products/enterprise/tde.html. Online; accessed 14-May-2019. (**Cited** on page 43.)

[Neu] Neuvonen, Simo and Wolski, Antoni and Manner, Markku and Raatikka, Vilho. Telecom Application Transaction Processing Benchmark (TATP). (**Cited** on page 72.)

[NKW15] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 644–655, New York, NY, USA, 2015. Association for Computing Machinery. (**Cited** on pages 14 and 47.)

[ora] Oracle Transparent Data Encryption (TDE) Technology. `https://docs.oracle.com/en/cloud/saas/marketing/responsys-user/EncryptionAtRest.htm` Online; accessed 14-May-2019. (**Cited** on page 43.)

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999. (**Cited** on page 30.)

[PBM+17] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery. (**Cited** on page 17.)

[PBP19] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: An Encrypted Database Using Semantically Secure Encryption. *Proc. VLDB Endow.*, 12(11):1664–1678, July 2019. (**Cited** on page 48.)

[PMPV16] Rogerio Pontes, Francisco Maia, João Paulo, and Ricardo Manuel Pereira Vilaça. SafeRegions: Performance Evaluation of Multi-party Protocols on HBase. In *35th IEEE Symposium on Reliable Distributed Systems Workshops, SRDS 2016 Workshop, Budapest, Hungary, September 26, 2016*, pages 31–36. IEEE Computer Society, 2016. (**Cited** on page 17.)

[PMVM19] Rogério Pontes, Francisco Maia, Ricardo Vilaça, and Nuno Machado. d'Artagnan: A Trusted NoSQL Database on Untrusted Clouds. In *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*, pages 61–70. IEEE, 2019. (**Cited** on page 17.)

[PPB+17] Rogério Pontes, Mário Pinto, Manuel Barbosa, Ricardo Vilaça, Miguel Matos, and Rui Oliveira. Performance Trade-Offs on a Secure Multi-Party Relational Database. In *Proceedings of the Symposium on Applied Computing*, SAC '17,

page 456–461, New York, NY, USA, 2017. Association for Computing Machinery. (**Cited** on page 18.)

[PPRY18] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with Logarithmic Overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882, Oct 2018. (**Cited** on pages 99 and 107.)

[PRZB11] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100. ACM, 2011. (**Cited** on pages 15, 47 and 77.)

[PVC18] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB – A Secure Database using SGX. In *IEEE Symposium on Security & Privacy, May 2018*. IEEE, May 2018. (**Cited** on page 50.)

[Red] Redis. Redis. (**Cited** on page 88.)

[RGR18] David Reinsel, John F Gantz, and John Rydning. The Digitization of the World From Edge to Core. *IDC Analyze the Future*, 2018. (**Cited** on page 12.)

[SB18] Hovav Shacham and Alexandra Boldyreva, editors. *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*. Springer, 2018. (**Cited** on page 89.)

[SCF+15] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, 2015. (**Cited** on page 103.)

[Sch93] Bruce Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *FSE*, 1993. (**Cited** on page 42.)

[SDS+18] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM*, 65(4):18:1–18:26, April 2018. (**Cited** on pages 99, 108, 111 and 116.)

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012. (**Cited** on page 41.)

[SGK+85] R. Sandberg, David Goldberg, Steven Lawrence Kleiman, Dan Walsh, and Robert Lyon. Design and Implementation of the Sun Network Filesystem. *USENIX*, pages 119–130, 1985. (**Cited** on page 42.)

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. (**Cited** on page 81.)

[SPS14] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, California, USA*. The Internet Society, 2014. (**Cited** on pages 46 and 99.)

[SS13] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267, May 2013. (**Cited** on page 117.)

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards Practical Oblivious RAM. In *NDSS*. The Internet Society, 2012. (**Cited** on page 115.)

[Sto10] Michael Stonebraker. SQL Databases v. NoSQL Databases. *Commun. ACM*, 53(4):10–11, April 2010. (**Cited** on page 13.)

[TBO+13] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 182–196, New York, NY, USA, 2013. Association for Computing Machinery. (**Cited** on pages 54 and 74.)

[TGS+15] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra Incognita: On the Practicality of User-Space File Systems. In *HotStorage*, 2015. (**Cited** on page 43.)

[TH20] Florian Kerschbaum Timon Hackenjos, Florian Hahn. SAGMA: Secure Aggregation Grouped by Multiple Attributes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, Portland, USA, June 14 - July 19, 2020*. ACM, 2020. (**Cited** on page 48.)

[TKMZ13] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5):289–300, March 2013. (**Cited** on page 47.)

[UAA+10] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Całkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System.

In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, page 17, USA, 2010. USENIX Association. (**Cited** on page 54.)

[UPvS09] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Comput. Netw.*, 53(11):1830–1845, July 2009. (**Cited** on page 73.)

[VSV12] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: a cloud-backed file system for the enterprise. In *FAST*, 2012. (**Cited** on page 43.)

[VTZ17] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 59–72, USA, 2017. USENIX Association. (**Cited** on page 54.)

[WAK18] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. Sgx-Perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 201–213, New York, NY, USA, 2018. Association for Computing Machinery. (**Cited** on page 104.)

[Wal95] Stephen R. Walli. The POSIX Family of Standards. *StandardView*, 3(1):11–17, March 1995. (**Cited** on page 53.)

[weba] Docker and AUFS in practice. `https://docs.docker.com/storage/storagedriver/aufs-driver/`. (**Cited** on page 67.)

[webb] FileBench. `https://github.com/filebench/filebench`. (**Cited** on page 67.)

[webc] Leveldb. `https://github.com/google/leveldb`. (**Cited** on page 67.)

[webd] OpenSSL. `https://www.openssl.org`. (**Cited** on page 63.)

[webe] S3FS. `https://github.com/s3fs-fuse/s3fs-fuse`. (**Cited** on page 54.)

[webf] SshFS. `https://github.com/libfuse/sshfs`. (**Cited** on page 54.)

[webg] Tuning KVM. `http://www.linux-kvm.org/page/Tuning_KVM`. (**Cited** on page 67.)

[Wic94] Stephen B. Wicker. *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994. (**Cited** on page 64.)

[WKC+14] Wai Kit Wong, Ben Kao, David Wai-Lok Cheung, Rongbin Li, and Siu-Ming Yiu. Secure query processing with data interoperability in a cloud database environment. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1395–1406. ACM, 2014. (**Cited** on page 49.)

[WMZ03a] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association. (**Cited** on pages 42 and 61.)

[WMZ03b] Charles P. Wright, Michael C. Martino, and Erez Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 197–210. USENIX, 2003. (**Cited** on page 60.)

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM. (**Cited** on pages 99, 100, 101, 108, 114 and 133.)

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 39–56, New York, NY, USA, 2017. ACM. (**Cited** on page 95.)

[XCP15] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015. (**Cited** on page 103.)

[YWW+16] Xingliang Yuan, Xinyu Wang, Cong Wang, Chen Qian, and Jianxiong Lin. Building an Encrypted, Distributed, and Searchable Key-value Store. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 547–558, 2016. (**Cited** on page 48.)

[ZABN01] Erez Zadok, Johan M. Andersen, Ion Badulescu, and Jason Nieh. Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 289–304, Berkeley, CA, USA, 2001. USENIX Association. (**Cited** on page 54.)

[ZBS98] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998. `www.cs.columbia.edu/~library`. (**Cited** on page 42.)

[ZDB+17] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design*

and Implementation (NSDI 17), pages 283–298, Boston, MA, March 2017. USENIX Association. (**Cited** on pages 50 and 99.)

[ZLP+17] Wenting Zheng, Frank H. Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. Minicrypt: Reconciling encryption and compression for big data stores. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 191–204. ACM, 2017. (**Cited** on page 48.)

[ZN00] Erez Zadok and Jason Nieh. FIST: A Language for Stackable File Systems. *SIGOPS Oper. Syst. Rev.*, 34(2):38, April 2000. (**Cited** on page 54.)