Secure and Efficient Processing of Outsourced Data Structures using Trusted Execution Environments

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Benny Fuhry, M. Sc.
aus Buchen (Odenwald)

Mannheim, 2020

Dekan: Dr. Bernd Lübcke, Universität Mannheim
Referent: Prof. Dr. Frederik Armknecht, Universität Mannheim
Korreferent: Prof. Dr. Florian Kerschbaum, University of Waterloo

Tag der mündlichen Prüfung: 12.01.2021

Abstract

In recent years, more and more companies make use of cloud computing; in other words, they outsource data storage and data processing to a third party, the cloud provider. From cloud computing, the companies expect, for example, cost reductions, fast deployment time, and improved security. However, security also presents a significant challenge as demonstrated by many cloud computing–related data breaches. Whether it is due to failing security measures, government interventions, or internal attackers, data leakages can have severe consequences, e.g., revenue loss, damage to brand reputation, and loss of intellectual property. A valid strategy to mitigate these consequences is data encryption during storage, transport, and processing. Nevertheless, the outsourced data processing should combine the following three properties: strong security, high efficiency, and arbitrary processing capabilities.

Many approaches for outsourced data processing based purely on cryptography are available. For instance, encrypted storage of outsourced data, property-preserving encryption, fully homomorphic encryption, searchable encryption, and functional encryption. However, all of these approaches fail in at least one of the three mentioned properties.

Besides approaches purely based on cryptography, some approaches use a *trusted execution environment* (TEE) to process data at a cloud provider. TEEs provide an isolated processing environment for user-defined code and data, i.e., the confidentiality and integrity of code and data processed in this environment are protected against other software and physical accesses. Additionally, TEEs promise efficient data processing.

Various research papers use TEEs to protect objects at different levels of granularity. On the one end of the range, TEEs can protect entire (legacy) applications. This approach facilitates the development effort for protected applications as it requires only minor changes. However, the downsides of this approach are that the attack surface is large, it is difficult to capture the exact leakage, and it might not even be possible as the isolated environment of commercially available TEEs is limited. On the other end of the range, TEEs can protect individual, stateless operations, which are called from otherwise unchanged applications. This approach does not suffer from the problems stated before, but it leaks the (encrypted) result of each operation and the detailed control flow through the application. It is difficult to capture the leakage of this approach, because it depends on the processed operation and the operation's location in the code.

In this dissertation, we propose a trade-off between both approaches: the TEE-based processing of data structures. In this approach, otherwise unchanged applications call a TEE for self-contained data structure operations and receive encrypted results. We examine three data structures: TEE-protected B^+-trees, TEE-protected database dictionaries, and TEE-protected file systems. Using these data structures, we design three secure and efficient systems: an outsourced system for index searches; an outsourced, dictionary-encoding–based, column-oriented, in-memory database supporting analytic queries on large datasets; and an outsourced system for group file sharing supporting large and dynamic groups.

Due to our approach, the systems have a small attack surface, a low likelihood of security-relevant bugs, and a data owner can easily perform a (formal) code verification of the sensitive code. At the same time, we prevent low-level leakage of individual operation results. For all systems, we present a thorough security evaluation showing lower bounds of security. Additionally, we use prototype implementations to present upper bounds on performance. For our implementations, we use a widely available TEE that has a limited isolated environment—Intel Software Guard Extensions. By comparing our systems to related work, we show that they provide a favorable trade-off regarding security and efficiency.

Zusammenfassung

Seit einigen Jahren nutzen immer mehr Unternehmen Cloud Computing. Hierbei wird die Datenspeicherung und -verarbeitung zu einer dritten Partei, dem sogenannten Cloud-Anbieter, ausgelagert. Von Cloud Computing versprechen sich Unternehmen unter anderem folgende Vorteile: reduzierte Kosten, schnellere Bereitstellung und erhöhte Sicherheit. Zahlreiche Datenpannen haben jedoch gezeigt, dass mangelnde Sicherheit noch immer ein großes Problem beim Cloud Computing darstellt. Für Unternehmen können solche Datenpannen verheerende Folgen haben, wie z. B. Umsatzeinbrüche, Reputationsverlust und Verlust von geistigem Eigentum. Diese negativen Konsequenzen können minimiert werden, wenn die ausgelagerten Daten während der Speicherung, Übertragung und Verarbeitung verschlüsselt sind. Neben der Sicherheit der Daten muss dabei stets eine hohe Effizienz sowie die uneingeschränkte Verarbeitung gewährleistet werden.

Bei folgenden, beispielhaft aufgeführten Ansätzen der ausgelagerten Datenverarbeitung beruht die Sicherheit nur auf Kryptographie: verschlüsseltes Auslagern von Daten, eigenschaftserhaltende Verschlüsselung (property-preserving encryption), vollhomomorphe Verschlüsselung (fully homomorphic encryption), durchsuchbare Verschlüsselung (searchable encryption) und funktionale Verschlüsselung (functional encryption). Keiner dieser Ansätze vereint die drei obengenannten Kriterien.

Daneben gibt es Ansätze, bei denen die Sicherheit neben Kryptographie auch auf Trusted Execution Environments (TEEs) beruht, die beim Cloud-Anbieter für die Datenverarbeitung eingesetzt werden. TEEs bieten eine isolierte Umgebung, die die Integrität und Vertraulichkeit der darin ausgeführten Programme und der verarbeiteten Daten gegenüber anderen Programmen und physischem Zugriff schützt. Außerdem ist die Effizienz der Datenverarbeitung hoch.

TEEs können Objekte von unterschiedlicher Granularität absichern: Auf der einen Seite können komplette Applikationen geschützt werden. Wenngleich dabei der Schutz (bestehender) Applikationen mit geringem Aufwand erfolgen kann, ist dennoch die Angriffsfläche groß und potenzielle Datenlecks sind schwer zu bestimmen. Da aktuell verfügbare TEEs eine beschränkte Größe der isolierten Umgebung haben, ist dieser Ansatz häufig nicht umsetzbar. Auf der anderen Seite können TEEs einzelne, zustandslose Operationen schützen, die dann von ansonsten unveränderten Applikationen aufgerufen werden. Dieser Ansatz leidet zwar nicht unter den zuvor beschriebenen Problemen, aber ein Angreifer kann das (verschlüsselte) Ergebnis jeder Operation und den genauen Programm-Ablauf lernen.

In dieser Dissertation wird ein Kompromiss aus den beiden zuvor beschriebenen Lösungen vorgeschlagen: die Nutzung von TEEs, um Datenstrukturen zu verarbeiten. Dabei verwenden ansonsten unveränderte Applikationen Datenstruktur-Operationen, die in sich abgeschlossen von einer TEE verarbeitet werden. Bestandteil der Untersuchung sind drei Datenstrukturen: TEE-geschützte B^+-Bäume, TEE-geschützte Datenbank-Wörterbücher und TEE-geschützte Dateisysteme. Mit diesen Datenstrukturen werden drei sichere und effiziente ausgelagerte Systeme entworfen: ein System für Index-Suchen, eine Wörterbuch-basierte, spaltenorientierte, In-Memory-Datenbank und ein System für den gruppenbasierten Austausch von Dateien.

Der in dieser Arbeit vorgestellte Ansatz führt dazu, dass die Systeme eine möglichst kleine Angriffsfläche und eine geringe Anfälligkeit für sicherheitsrelevante Programmierfehler haben. Der sicherheitskritische Programmcode ist einfach zu überprüfen und ein Angreifer kann das Ergebnis einzelner Operationen nicht lernen. Ausführliche Sicherheitsevaluierungen zeigen untere Sicherheitsschranken der Systeme und prototypische Implementierungen werden genutzt, um obere Schranken der Performanz aufzuzeigen. Hierbei kommt Intel Software Guard Extensions, eine weit verbreitete TEE mit beschränkter Größe der isolierten Umgebung, zum Einsatz.

Danksagung

Ich danke Florian Kerschbaum, der mir dieses praktisch-orientierte Thema vorgeschlagen und die Promotion durch seine harte Schule zum Erfolg geführt hat. Von der Problemsuche über das Design, die Implementierung und die Evaluierung bis zum Aufschrieb hat er mich fachlich unterstützt und mir sehr viel beigebracht. Daneben möchte ich mich insbesondere bei Frederik Armknecht bedanken, der mir eine Promotion an der Universität Mannheim ermöglicht und mir wertvolles Feedback gegeben hat.

Den SAP-Managern, die ich über die Jahre hatte – Andreas Schaad, Roger Gutbrod, Detlef Plümper und Mathias Kohler – gilt ebenso mein Dank. Sie haben mir die notwendige Freiheit gegeben, diese Dissertation zu schreiben, und mir vieles vermittelt, das für meine berufliche Laufbahn von unschätzbarem Wert ist. Ein weiterer Dank gilt allen Teamkollegen und Studenten, die mich bei der Dissertation unterstützt haben. Besonders danke ich den Krypto-Kollegen: Hahn, der mich auf diese wahnwitzige Idee gebracht hat und diese dann ausbaden musste, Andreas, der mir selbst die absurdesten Auswüchse von Git erklären konnte und Anselme, der auch beim größten Chaos immer gelassen blieb. Die Anonymisierungs-Kollegen Daniel, Jonas und Benjamin dürfen nicht unerwähnt bleiben, haben wir uns doch gemeinsam durch das Dickicht der Spitzenforschung geschlagen. Tom gilt ein besonderer Dank, da er sich durch die unzähligen Seiten meiner Dissertation quälte, um mein Englisch auszubessern.

Ein weiterer Dank gilt Jürgen Sillmann, der mein Interesse an Computern in jungen Jahren geweckt hat. Joachim Müller gebührt ebenfalls mein Dank, da er mich bei den ersten Schritten in der Programmierung begleitete. Dankbar bin ich ihm auch für die Hinweise, dass korrekte Rechtschreibung und Grammatik nicht optional sind und dass ich ohne Englisch nicht sehr weit kommen werde.

Meinen Eltern, Hanne und Wilfried, danke ich dafür, dass sie mir meine akademische Ausbildung ermöglicht und meine Entscheidungen immer unterstützt haben. Ihre Erziehung hat mich gelehrt, Probleme jeglicher Art zu erkennen, sie zu lösen und auch dann nicht aufzugeben, wenn es schwierig wird. Für unzählige positive Erlebnisse danke ich meinen Geschwistern Conny, Sandra und Dirk. Durch sie bin ich zu dem Menschen geworden, der ich bin und der diese Dissertation verfasst hat. Dies gilt ebenfalls für meine langjährigen Freunde Schell und Flo, die zusätzlich für notwendige Ablenkung gesorgt haben.

Abschließend danke ich meiner Freundin Nanni, dir mir bereits seit über 15 Jahren die Kraft gibt, alles zu meistern, was ich mir vornehme. Ohne sie an meiner Seite wäre diese Dissertation nicht möglich gewesen.

Contents

Abbreviations

ABE attribute-based encryption
AE authenticated encryption
AES advanced encryption standard
AMD SEV AMD Secure Encrypted Virtualization
AMD SEV-SNP AMD SEV-Secure Nested Paging

BE broadcast encryption
BW business warehouse

CA certificate authority
CSR certificate signing request

DBMS database management system
DET deterministic encryption
DoS denial of service

ECall enclave call
EPC enclave page cache
EPCM enclave page cache map

FE functional encryption
FHE fully homomorphic encryption
FPGA field programmable gate array

HE hybrid encryption
HSM hardware security module

IBBE identity-based broadcast encryption
IBE identity-based encryption
IBM SE IBM Secure Execution
IND-CCA indistinguishability under chosen ciphertext attacks
IND-CPA indistinguishability under chosen plaintext attacks
Intel SGX Intel Software Guard Extensions
Intel SGX PCL Intel SGX Protected Code Loader

LE launch enclave
LOC lines of code

MAC message authentication code
MOPE modular OPE
MPC secure multi-party computation

OCall outside call
OLAP online analytical processing
OLTP online transaction processing
OPE order-preserving encryption
ORAM oblivious RAM
ORE order-revealing encryption
OS operating system

PE provisioning enclave
PHE partial homomorphic encryption
PKI public key infrastructure
PPE property-preserving encryption
PPT probabilistic polynomial-time
PRF pseudorandom function
PRM processor reserved memory
PRP pseudorandom permutation

QE quoting enclave

ROP return-oriented programming
RPE range predicate encryption

SDK software development kit
SE searchable encryption
SKE symmetric-key encryption
SSE symmetric searchable encryption
SVN security version number

TCB trusted computing base
TEE trusted execution environment
TPM trusted platform module

1 Introduction

For decades, companies owned, managed, and maintained their applications and infrastructure. In 1996, the term "cloud computing" was coined [1] referring to a concept in which companies outsource their applications and infrastructure to a third party. This third party is called the *cloud provider* and it provides its services over a network; in most cases, the Internet. In 2006, cloud computing gained more attention as large companies, e.g., Amazon and Google, used the term and began to offer commercially available cloud computing services.

A recent cloud computing survey [2] states that, in 2018, 73% of the respondents used cloud-based applications and infrastructure, and the usage increased to 81% in 2020. Furthermore, 2% of the survey respondents plan to adopt cloud applications in the next 12 months and an additional 6% plan adoption in the next 12 to 36 months. In 2021, companies plan to invest 32% of their total IT budget into cloud computing. This trend is visible across many industries, e.g., education, manufacturing, healthcare, financial service, and government.

The top five reasons for companies to outsource applications and infrastructure to the cloud are the following: cost reduction; faster deployment time; increased efficiency; improved security; and increased flexibility and choice [3]. In contrast, the top three challenges with cloud computing are the following: controlling cloud costs; data privacy and security; and securing/protecting cloud resources [2]. Thus, on the one hand, companies expect an increased security from outsourced data processing, which is a valid assumption according to Gartner [4]. On the other hand, companies fear to lose their sensitive and business-critical data, which is also a valid fear considering recent cloud-related data breaches [5]–[8]. According to a recent IDC survey of 300 chief information security officers [9], 79% of the participating companies report a cloud data breach in the last 18 months.

Multiple security measures are recommended to mitigate the risk of data breaches, e.g., network firewalls, hardened *operating systems* (OSes), and virus scanners. However, these measures can have bugs or can be misconfigured. Additionally, governments might subpoena the cloud providers to hand over data, which circumvents these measures. Even if all security measures are successful and the cloud provider does not cooperate with governments, the *data owner* might not trust the cloud provider, because an internal attacker can easily leak sensitive data.

Whether it is due to failing security measures, governments, or internal attackers, a data leak can have severe consequences, e.g., revenue loss, damage to brand reputation, and loss of intellectual property. To mitigate the consequences of a data leak, data owners should incorporate data encryption as a last line of defense throughout the data's lifecycle, i.e., during storage, transport, and processing. We denote this concept by *secure, outsourced data processing*.

Ideally, outsourced data processing approaches combine strong security, high efficiency, and arbitrary processing capabilities. Many approaches in the literature, such as the following three, are based purely on cryptography:

1. In the *encrypted outsourced storage* approach, the data owner encrypts its data locally with a secure encryption scheme, e.g., *advanced encryption standard* (AES), outsources

the data, and retains the encryption key. This approach provides perfect security, but the cloud provider cannot do any processing. Consequently, the efficiency is low as all data needs to be downloaded for local processing at the data owner.

2. *Property-preserving encryption* (PPE) [10]–[12] preserves properties of the underlying plaintext data in the corresponding ciphertexts. As a result, PPE allows some processing capabilities on encrypted data, e.g., equality comparisons and range queries. The efficiency for these operations is high, but the security is debatable [13]–[15].
3. *Fully homomorphic encryption* (FHE) [16]–[18] enables the processing of arbitrary circuits on encrypted data and offers semantic security. However, it is too inefficient for adoption in practical systems [19].

Alternatively, a *trusted execution environment* (TEE) at the cloud provider can support the secure, outsourced data processing. TEEs provide an isolated processing environment for user-defined code and data, i.e., the confidentiality and integrity of code and data processed in this environment are protected against other software and physical accesses. In particular, a TEE protects against all privileged software, e.g., OS, hypervisor, and firmware. Furthermore, the processing of protected data is very efficient. Thus, TEEs are ideally suited for secure and efficient data processing in a scenario with an untrusted cloud provider.

In this dissertation, we use a TEE to protect the outsourced processing of data structures. We embed such TEE-protected data structures in cloud-based systems and show that this approach achieves strong security, high efficiency, and arbitrary processing capabilities.

In the remainder of this chapter, we first describe the contributions of this dissertation in Section 1.1. Then, in Section 1.2, we present the outline of the dissertation and a reading guideline. We conclude the chapter in Section 1.3 with the publications related to this dissertation.

1.1 Contributions

The research question we address in this dissertation is the following:

> **For outsourced systems using a memory-limited, widely available trusted execution environment (TEE) to process data structures at an untrusted cloud provider, what are lower bounds of security and corresponding upper bounds on performance?**

This research question specifies the target technology, the scenario, the main investigation object—data structures, and the security and performance goals of this dissertation. In the following, we elaborate on these four points and then describe the main contributions of this dissertation.

Target technology. We assume that a cloud provider offers TEE-enabled machines as part of its cloud computing service. Such a TEE provides an isolated processing environment for user-defined code, a so-called *enclave*, which securely processes data. Besides confidentiality and integrity protection for user-defined enclaves, we require that the used TEE supports at least four capabilities: (1) remote attestation, i.e., the enclave can prove its integrity and that it is protected by a specific TEE to a remote party; (2) remote provisioning, i.e., a remote party can securely provision sensitive data into the enclave; (3) data sealing, i.e., the enclave can store data in the untrusted environment while the TEE guarantees the data's confidentiality, integrity, and freshness; and (4) source of randomness, i.e., the enclave can securely access a trusted source of randomness.

In 2015, Intel released CPUs supporting *Intel Software Guard Extensions* (Intel SGX) [20]–[28], a TEE which is now widely available in most Intel Core and Xeon processors. During the timeframe in which the research for this dissertation was done, Intel SGX was the only TEE providing the capabilities listed before[1]. Therefore, we use Intel SGX as TEE throughout this dissertation. Intel SGX enables arbitrary processing of encrypted data, it is highly efficient, and the concept is secure.

A downside of Intel SGX is that it is memory limited, i.e., the enclave code and the data processed by the enclave *at once* cannot exceed 96 MB without a severe performance decrease. As other papers [29]–[31] and this dissertation show, Intel SGX still enables secure, outsourced processing of data far beyond 96 MB, and Intel SGX's efficiency is vastly superior to related approaches without a TEE. Another downside is that various attacks on Intel SGX were presented over the last years. The attacks result from side channels [32]–[34], enclave code exploits [35], [36], enclave-based malware [37], [38], and processor bugs [39]–[41]. In this dissertation, we argue that the main strategy to mitigate attacks on Intel SGX is to have a small enclave size, i.e., the enclave should only have a few *lines of code* (LOC).

Scenario. As mentioned in the introduction, data owners should incorporate encryption throughout the lifecycle of their sensitive, outsourced data, and we call this concept secure, outsourced data processing. As we assume that the cloud provider uses a TEE for data processing, we denote the scenario considered in the dissertation by *secure, outsourced, TEE-based data processing*. This scenario is illustrated in Figure 1.1a and it works as follows: A trusted data owner encrypts its data locally and outsources the encrypted data to an untrusted cloud provider. Whenever the data owner wants to process the outsourced data, it sends a corresponding request, which might contain encrypted data, to the cloud provider. The data is processed by the cloud provider using a TEE and a response is sent back to the data owner, who might need to decrypt contained data. In this scenario, we interchangeably use the terms *user* and data owner, as the data owner outsources and uses the outsourced data. The user can also be an application, which uses the outsourced processing capabilities. A real-world example for this scenario is the following: a company outsources its database; an application of the same company triggers database requests and receives the corresponding result sets.

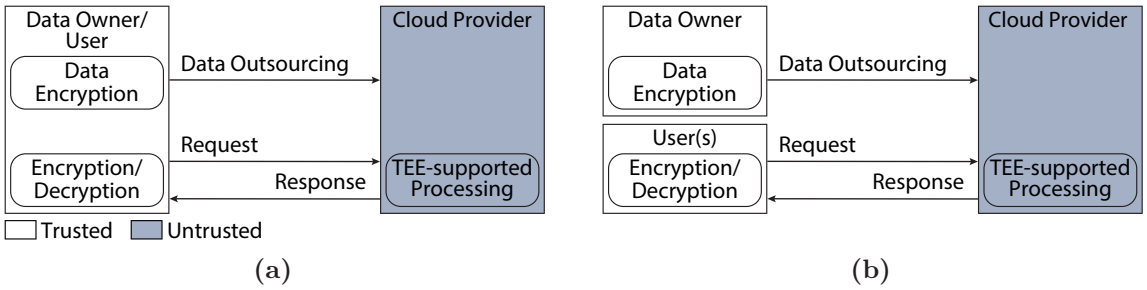

Figure 1.1: Secure, outsourced, TEE-based data processing scenario variants: (a) data owner and user are equal, (b) data owner and user(s) are independent.

In some cases, we consider a variant of this scenario, which is presented in Figure 1.1b. The only difference is that the data owner and the user(s) are distinct. A real-world example for this variant is the following: a student outsources her files to the cloud and other students can request (parts of) these files.

Data structures. The secure, outsourced, TEE-based data processing scenario does not define the processing done inside an enclave, which ranges from the executing of entire applications

1 We discuss alternative TEEs and their capabilities in Section 3.3.

to individual, stateless operations. Both extremes have their advantages and disadvantages (see Section 4.8). In this dissertation, we use a trade-off and process data structures inside an enclave.

In particular, we examine the secure, outsourced, TEE-based data processing of three data structures in the three main chapters of this dissertation: B^+-trees, database dictionaries, and file systems. A B^+-tree is a balanced, n-ary search tree, and our outsourced B^+-trees can be used to search single search keys and search key ranges. A database dictionary provides data compression in a column-oriented database, and our outsourced dictionaries enable equality and range searches. A file system contains files and directories, and our outsourced file system allows, among other features, remote file storage, group data sharing, and deduplication.

Security and performance goals. In this dissertation, we intend to enhance the architecture of outsourced systems using TEE-protected data structures. Our goal is to achieve a trade-off that improves over existing work regarding security and performance. In more detail, the outsourced data processing systems should:

- Limit the amount of leaked information in transit between the user and cloud provider; during storage at the cloud provider; and during processing at the cloud provider.
- Provide an as low as possible latency to the user of the outsourced processing.

Main contributions. In the secure, outsourced, TEE-based data processing scenario, the TEE can protect different objects. The proposition of this dissertation is that TEEs should be used to process outsourced data structures. From this proposition, we expect a much smaller enclave size compared to approaches which protect entire applications with a TEEs. A small enclave size has multiple benefits, e.g., a small attack surface, a low likelihood of bugs, a low interface complexity, and a facilitation for code verification by the data owner. Additionally, we expect to not leak low-level result as done by approaches which protect individual, stateless operations with a TEEs.

To verify our proposition, we design three TEE-protected data structures and use them to build three systems:

- Using TEE-protected B^+-trees, we design an outsourced system for index searches—denoted by HardIDX.
- Using TEE-protected dictionaries, we design an outsourced, dictionary-encoding–based, column-oriented, in-memory database supporting analytic queries on large datasets—denoted by EncDBDB.
- Using a TEE-protected file system, we design an outsourced system for group file sharing supporting large and dynamic groups—denoted by SeGShare.

For all three systems, we present a thorough security evaluation showing lower bounds of security. Additionally, we use Intel SGX, a memory-limited, widely available TEE for prototype implementations. Based on these implementations, we present upper bounds on performance. Consequently, we use three examples to provide an answer to the research question stated at the beginning of this section.

1.2 Outline

In the following, we provide the outline of this dissertation. At the end of this section, we provide a brief reading guideline.

In **Chapter 2**, we first explain the notation that is used throughout this dissertation. Then, we formally introduce the cryptographic primitives that are used by our secure data structures; by related, secure, outsourced data processing approaches; and by various TEEs.

In **Chapter 3**, we provide definitions for TEEs, enclaves, the enclave memory, and the enclave size. Additionally, we list TEE capabilities that we require for the approaches presented in this dissertation. Then, we present details of the only TEE that fulfills these capabilities—Intel SGX. As many news articles about Intel SGX cover attacks on it, we dedicate a section to attacks on Intel SGX and mitigations against these attacks. Finally, we differentiate technologies related to TEEs and classify commercially available TEEs according to our list of required capabilities.

In **Chapter 4**, we differentiate this dissertation from related approaches, i.e., approaches that also enable secure, outsourced data processing (with and without a TEE). We show that all of these approaches cover another scenario and/or they fail in at least one of the following aspects: strong security, high efficiency, or arbitrary processing capabilities. Namely, we describe *encrypted outsourced storage*, *secure multi-party computation* (MPC), *fragmentation*, *property-preserving encryption* (PPE), *searchable encryption* (SE), *homomorphic encryption*, *functional encryption* (FE), and TEE-based approaches that do not process a data structure in an enclave.

In **Chapter 5**, we motivate and describe the design principles that we use for our TEE-protected data structures. Additionally, we present the security and performance assessment methodology that we use to answer our research question.

Chapters 6, 7, and 8 are the three main chapters of this dissertation. In these chapters, we present a system for secure index searches using TEE-protected B^+-trees, a secure database using TEE-protected database dictionaries, and a system for group file sharing using a TEE-protected file system, respectively. Each of the three chapters follows the same schema: In the introduction, we outline the TEE-protected data structure covered in the chapter, introduce the system that uses this data structure, mention related approaches, and present a list of contributions of the chapter. In a following design considerations section, we provide details about the data structure (without protection), the system, and the attacker model. Afterwards, we differentiate our system from existing approaches in a related work section. At this point in each chapter, the foundations are expressed, and we proceed to the design of our TEE-protected data structure and the corresponding system. To provide a concise description in the design section, we extract possible extensions to a separate section that follows after the design section. Then, we briefly discuss the implementation of the system in a dedicated section. The following evaluation section uses the implementation for a thorough performance (and storage overhead) evaluation. Besides, the evaluation section provides a security evaluation. Each main chapter concludes with a brief summary.

Chapter 9 concludes this dissertation. We first summarize all chapters and relate them to our research questions. Then, we give an outlook on potential future research avenues.

Reading guideline. As we illustrate in Figure 1.2, the main reading path of this dissertation is to read through all chapters subsequently. However, Chapter 3 might be of interest to a reader that wants to learn details about Intel SGX and related TEEs, independent of the remainder of this dissertation. If the reader is familiar with cryptographic primitives and Intel SGX, Chapter 6, 7, and 8 are understandable without the preceding chapters and the chapters are independent of each other.

1.3 Related Publications

Ideas developed in the process of writing this dissertation have been published or submitted. The following publications are directly related and are used in the specified chapters:

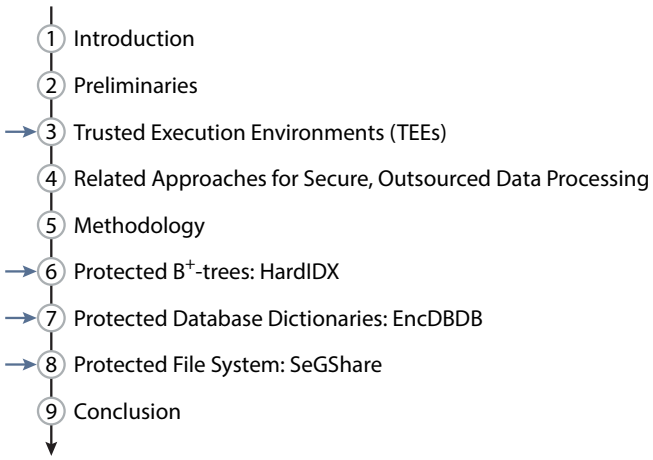

Figure 1.2: Reading guideline for this dissertation.

- B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "HardIDX: Practical and Secure Index with SGX", in *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*, ser. DBSec, 2017.

  This publication forms the basis for Chapter 6 covering outsourced B^+-trees. It received the best paper award at the DBSec 2017.
- B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "HardIDX: Practical and secure index with SGX in a malicious environment", *Journal of Computer Security*, JCS, 2018.

  This publication extends the last publication by considering an outsourced B^+-tree secure against a malicious cloud attacker. This extension is also part of Chapter 6.
- B. Fuhry, J. Jayanth H A, and F. Kerschbaum, "EncDBDB: Searchable Encrypted, Fast, Compressed, In-Memory Database using Enclaves", arXiv.org, arXiv:2002.05097, 2020.

  This publication forms the basis for Chapter 7 covering outsourced database dictionaries, which are used in column-oriented databases for data compression.
- B. Fuhry, L. Hirschhoff, S. Koesnadi, and F. Kerschbaum, "SeGShare: Secure Group File Sharing in the Cloud using Enclaves", in *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2020.

  This publication forms the basis for Chapter 8 covering outsourced file systems.

The following paper gained from the insights of this dissertation, but its content is not part of the dissertation:

- A. Fischer, B. Fuhry, F. Kerschbaum, and E. Bodden, "Computation on Encrypted Data using Dataflow Authentication", *Proceedings on Privacy Enhancing Technologies*, PoPETS, 2020.
- A. Fischer, B. Fuhry, J. Kussmaul, J. Janneck, F. Kerschbaum, and E. Bodden, "Improved Computation on Encrypted Data using Dataflow Authentication", (under submission), 2020.

2 Preliminaries

In this chapter, we first review the notation that is used throughout this dissertation, in Section 2.1. Afterwards, in Section 2.2, we formally introduce the cryptographic primitives that are used by the secure data structures presented in this dissertation, by related secure, outsourced data processing approaches, and by various TEEs. Namely, we describe the following primitives: *pseudorandom functions* (PRFs), *pseudorandom permutations* (PRPs), *symmetric-key encryptions* (SKEs), *message authentication codes* (MACs), *authenticated encryptions* (AEs), *cryptographic hash functions*, *set hash functions*, and *Merkle trees*.

2.1 Notation

General notation.

- The set of positive integers is denoted by $\mathbb{N}$.
- The set of binary strings of length $n \in \mathbb{N}$ is denoted by $\{0,1\}^n$.
- The set of finite-length binary strings is denoted by $\{0,1\}^*$.
- The length of a binary string V in bits is denoted by $|V|$.
- The concatenation of two binary strings $U \in \{0,1\}^n$, $V \in \{0,1\}^m$ is denoted by $U||V \in \{0,1\}^{n+m}$.
- The exclusive or operation of two binary values U and V is denoted by $U \oplus V$.
- We use bold letters to refer value collections.
- A set $\boldsymbol{X}$ contains $|\boldsymbol{X}|$ unique values, i.e., $\boldsymbol{X} = \{X_0, \ldots, X_{|\boldsymbol{X}|-1}\}$ and $\forall\, i,j \in [0, |\boldsymbol{X}| - 1] \wedge i \neq j : X_i \neq X_j$. An empty set is denoted by $\emptyset$.
- A tuple $\boldsymbol{T}$ contains $|\boldsymbol{T}|$ values, i.e., $\boldsymbol{T} = (T_0, \ldots, T_{|\boldsymbol{T}|-1})$. An empty tuple is denoted by $\emptyset$.
- For a partially ordered set $\boldsymbol{X}$, $n, m, x \in \boldsymbol{X}$, and $n \leq m$, $\boldsymbol{R} = [n,m]$ denotes the range $\boldsymbol{R} = \{x \,|\, n \leq x \leq m\}$, $\boldsymbol{R} = [n,m)$ denotes the range $\boldsymbol{R} = \{x \,|\, n \leq x < m\}$, $\boldsymbol{R} = (n,m]$ denotes the range $\boldsymbol{R} = \{x \,|\, n < x \leq m\}$, and $\boldsymbol{R} = (n,m)$ denotes the range $\boldsymbol{R} = \{x \,|\, n < x < m\}$.
- A value V that is contained in a set $\boldsymbol{X}$, tuple $\boldsymbol{T}$, or range $\boldsymbol{R}$ is denoted by $V \in \boldsymbol{X}$, $V \in \boldsymbol{T}$, and $V \in \boldsymbol{R}$.
- A value V that is chosen uniformly at random from a set $\boldsymbol{X}$, tuple $\boldsymbol{T}$, or range $\boldsymbol{R}$ is denoted by $V \xleftarrow{\$} \boldsymbol{X}$, $V \xleftarrow{\$} \boldsymbol{T}$, and $V \xleftarrow{\$} \boldsymbol{R}$.
- For a named value, set, tuple, or range, we use superscript, e.g., V^{name}, $\boldsymbol{X}^{name}$, $\boldsymbol{T}^{name}$, and $\boldsymbol{R}^{name}$.
- If a value V has an attribute U, we denote it by $V.U$, and if V's attribute is a tuple $\boldsymbol{T}$ by $V.\boldsymbol{T}$.

Algorithms.

- We do not differentiate between a function and an algorithm, and use the terms interchangeably.

- The security parameter of algorithms is always denoted by $\lambda \in \mathbb{N}$. The unary representation 1^λ is used if it is passed to an algorithm.
- For a (probabilistic) algorithm A, $x \leftarrow \mathsf{A}$ denotes that A outputs x.
- For a tuple, we define an append function that appends a value to the end of the tuple, a pop function that returns the first value and removes it from the tuple, and a last function that returns the last value of the tuple. For instance, let $\boldsymbol{T} = \{T_0, T_1\}$ be a tuple, then $\boldsymbol{T}.\mathsf{Append}(T_2) = \{T_0, T_1, T_2\}$, $T_0 \leftarrow \boldsymbol{T}.\mathsf{Pop}()$ with $\boldsymbol{T} = \{T_1\}$ after the operation, and $T_1 \leftarrow \boldsymbol{T}.\mathsf{Last}()$ with $\boldsymbol{T} = \{T_0, T_1\}$ after the operation.

2.2 Cryptographic Primitives

In this section, we introduce cryptographic primitives that are relevant for this dissertation. Unless explicitly noted, the definitions are based on the textbook by Katz *et al.* [48]. We follow the asymptotic security approach stating that a scheme should be secure against an "efficient" adversary, which can break the security with a "very small probability". We call an adversary efficient if its computation is *probabilistic polynomial-time* (PPT), and the adversary's success probability is very small if it is negligible in the security parameter. These concepts are defined in the following:

Definition 1 (Probabilistic polynomial time (PPT))**.** *An algorithm* A *is* probabilistic *if it has access to a source of randomness, and it runs in* polynomial time *if there exists a polynomial* $p(\cdot)$ *such that, for every input* $V \in \{0,1\}^*$*, the computation* $\mathsf{A}(V)$ *terminates within* $p(|V|)$ *steps.*

Definition 2 (Negligible function)**.** *A function* f *is called* negligible *if for every polynomial* $p(\cdot)$ *there exists an* $N \in \mathbb{N}$ *such that for all* $n \in \mathbb{N}$ *with* $n > N$ *it holds that* $f(n) < \frac{1}{p(n)}$*. We denote a negligible function by* negl*.*

To model a powerful PPT adversary $\mathcal{A}$, the adversary may receive access to an oracle $\mathcal{O}$. This oracle computes an auxiliary function that $\mathcal{A}$ cannot compute (efficiently). We denote an adversary $\mathcal{A}$ with access to an oracle $\mathcal{O}$ by $\mathcal{A}^{\mathcal{O}(\cdot)}$.

2.2.1 Pseudorandom Function (PRF)

The definition of a *pseudorandom function* (PRF) requires the definition of a keyed function:

Definition 3 (Keyed function)**.** *A* keyed function *is a function* $\mathsf{F} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ *where the first input is called* key *denoted by* k*.*

A keyed function F is called efficient if a PPT algorithm can compute $\mathsf{F}(k, x)$. Furthermore, F is called *length-preserving* if the key, input, and output have the same length, i.e., $|k| = |x| = |\mathsf{F}(k, x)|$.

Definition 4 (*Pseudorandom function* (PRF))**.** *Let* $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ *be an efficient, length-preserving, keyed function. The function* F *is a* pseudorandom function *if for all probabilistic polynomial-time distinguishers* $\mathcal{D}$*, there exists a negligible function* negl *such that:*

$$\left|\mathsf{Pr}[\mathcal{D}^{F(k,\cdot)}(1^\lambda) = 1] - \mathsf{Pr}[\mathcal{D}^{f_n(\cdot)}(1^\lambda)]\right| = \mathsf{negl}(\lambda)$$

where $k \leftarrow \{0,1\}^\lambda$ *is chosen uniformly at random and* f_n *is a function chosen randomly from the set of all functions mapping* n*-bit strings to* n*-bit strings.*

2.2.2 Pseudorandom Permutation (PRP)

The definition of a *pseudorandom permutation* (PRP) requires the definition of a keyed permutation, which is based on a keyed function:

Definition 5 (Keyed permutation)**.** *Let* F *be a keyed function. The function* F *is a* keyed permutation *if it is bijection for every* k*.*

A keyed permutation F is efficient if a PPT algorithm can compute $\mathsf{F}(k,x)$ and the inverse $\mathsf{F}^{-1}(k,y)$. It is called length-preserving if $|\mathsf{F}(k,x)| = |x| = |k|$.

Definition 6 (*Pseudorandom permutation* (PRP))**.** *Let* $\mathsf{PRP} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ *be an efficient, length-preserving, keyed permutation. The function* PRP *is a* pseudorandom permutation *if for all probabilistic polynomial-time distinguishers* $\mathcal{D}$*, there exists a negligible function* negl *such that:*

$$\left|\mathsf{Pr}[\mathcal{D}^{\mathsf{F}(k,\cdot),\mathsf{F}^{-1}(k,\cdot)}(1^\lambda) = 1] - \mathsf{Pr}[\mathcal{D}^{f_n(\cdot),f_n^{-1}(\cdot)}(1^\lambda)]\right| = \mathsf{negl}(\lambda)$$

where $k \leftarrow \{0,1\}^\lambda$ *is chosen uniformly at random and* f_n *is a function chosen randomly from the set of all functions mapping* n*-bit strings to* n*-bit strings.*

2.2.3 Symmetric-Key Encryption (SKE)

A *symmetric-key encryption* (SKE) scheme uses a single secret key to encrypt and decrypt values. SKE schemes are useful in a single-party or two-party setting: If a single party does the encryption and decryption, no further setup is necessary. For instance, the party can encrypt a value, store the encrypted value, and later decrypt the value before reading. If the encryption and decryption party differ, a secure key distribution scheme is necessary to establish a shared key. After a successful key distribution, the encryption party can encrypt a value under the shared key. Then, the encryption party sends the encrypted value to the decryption party, which decrypts the value using the shared key.

The syntax of an SKE scheme is as follows:

Definition 7 (*Symmetric-key encryption* (SKE) syntax)**.** *A* symmetric-key encryption *scheme is a tuple of three PPT algorithms* $\mathsf{SKE} = (\mathsf{SKE\_Gen}, \mathsf{SKE\_Enc}, \mathsf{SKE\_Dec})$ *such that:*

$SK \leftarrow \mathsf{SKE\_Gen}(1^\lambda)$: *Take a security parameter* λ *as input and output a secret key* SK*.*

$C \leftarrow \mathsf{SKE\_Enc}(SK, V)$: *Take a secret key* SK *and a plaintext value* $V \in \{0,1\}^*$ *as input. Output the ciphertext* C*.*

$V \leftarrow \mathsf{SKE\_Dec}(SK, C)$: *Take a secret key* SK *and a ciphertext* C *as input. Return* V *iff* V *was encrypted with* $\mathsf{SKE\_Enc}$ *under the key* SK*. Otherwise, return* $\perp$*.*

For a meaningful definition, we require the correctness of an SKE scheme:

Definition 8 (SKE correctness)**.** *Let* SKE *denote a symmetric-key encryption scheme consisting of the three algorithms as in Definition 7. For every* λ*, every secret key* SK *output by* $\mathsf{SKE\_Gen}$*, and every* $V \in \{0,1\}^*$*, it holds that* $\mathsf{SKE\_Dec}(SK, \mathsf{SKE\_Enc}(SK, V)) = V$*.*

To define the security of an SKE scheme, it is necessary to define the threat model and the security goal, i.e., the power of an adversary and when a scheme is considered broken. In cryptography literature, it is common to define the threat model via a security experiment conducted between a challenger and an adversary, and the security goal by a probability with which the adversary succeeds in this experiment. For SKE schemes there are two widely accepted security definitions: *indistinguishability under chosen plaintext attacks* (IND-CPA)

and *indistinguishability under chosen ciphertext attacks* (IND-CCA). In the following, we recapitulate the IND-CCA experiment, the IND-CCA–security definition, and briefly mention the difference to the weaker IND-CPA–security definition.

Definition 9 (IND-CCA experiment)**.** *Let* SKE *denote a symmetric-key encryption scheme consisting of the three algorithms as in Definition 7. For an adversary* $\mathcal{A}$ *and a security parameter* λ*, the* IND-CCA experiment $\mathsf{Exp}^{IND-CCA}_{\mathcal{A},\mathsf{SKE}}$ *is defined as:*

1. *The challenger generates a secret key* $SK \leftarrow \mathsf{SKE\_Gen}(1^\lambda)$.
2. *The adversary* $\mathcal{A}^{\mathsf{SKE\_Enc}(SK,\cdot),\mathsf{SKE\_Dec}(SK,\cdot)}$ *receives input* 1^λ*, has access to an encryption oracle* $\mathsf{SKE\_Enc}(SK,\cdot)$ *and a decryption oracle* $\mathsf{SKE\_Dec}(SK,\cdot)$*, and outputs two challenge values* $V_0 \in \{0,1\}^n$ *and* $V_1 \in \{0,1\}^n$ *of the same length* n.
3. *The challenger choses a random bit* $b \xleftarrow{\$} \{0,1\}$*, computes* $C \leftarrow \mathsf{SKE\_Enc}(SK, V_b)$*, and gives* C *to* $\mathcal{A}^{\mathsf{SKE\_Enc}(SK,\cdot),\mathsf{SKE\_Dec}(SK,\cdot)}$.
4. *The adversary* $\mathcal{A}^{\mathsf{SKE\_Enc}(SK,\cdot),\mathsf{SKE\_Dec}(SK,\cdot)}$ *can further use his oracle accesses, but is not allowed to use the decryption oracle* $\mathsf{SKE\_Dec}(SK, \cdot)$ *on* C*. Eventually, the adversary outputs a bit* b'.
5. *The experiment outputs 1 iff* $b = b'$. $\mathcal{A}^{\mathsf{SKE\_Enc}(SK,\cdot),\mathsf{SKE\_Dec}(SK,\cdot)}$ *succeeds if the experiment outputs 1.*

Definition 10 (IND-CCA security)**.** *Let* SKE *denote a symmetric-key encryption scheme consisting of the three algorithms as in Definition 7 and* λ *its security parameter.* SKE *is* IND-CCA secure *if for all PPT adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$|\mathsf{Pr}[\mathsf{Exp}^{IND-CCA}_{\mathcal{A},\mathsf{SKE}} = 1]| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

The definition of IND-CPA security is very similar. The only difference is that the adversary in the corresponding experiment does not have access to a decryption oracle. Thus, it is a weaker attacker model.

2.2.4 Message Authentication Code (MAC)

A *message authentication code* (MAC) is used to protect the integrity of a value, i.e., prevent an adversary from modifying the value and confirm that the value originated from a key holder. MACs require a shared secret between the communicating parties, and we consider the symmetric-key setting. The sender uses the secret key to calculate an authentication tag t for the value V and sends (V, t) to the second party. The integrity of V is verified by the second party using the authentication tag t and the shared secret key. More formally:

Definition 11 (*Message authentication code* (MAC) syntax)**.** *A* message authentication code *scheme is a tuple of three PPT algorithms* $\mathsf{MAC} = (\mathsf{MAC\_Gen}, \mathsf{MAC\_Tag}, \mathsf{MAC\_Vrfy})$ *such that:*

$SK \leftarrow \mathsf{MAC\_Gen}(1^\lambda)$: *Take a security parameter* λ *as input and output a secret key* SK *with* $|SK| \geq \lambda$.

$t \leftarrow \mathsf{MAC\_Tag}(SK, V)$: *Take a secret key* SK *and a plaintext value* $V \in \{0,1\}^*$ *as input. Output the authentication tag* t.

$b \leftarrow \mathsf{MAC\_Vrfy}(SK, V, t)$: *Take a secret key* SK*, a value* V*, and an authentication tag* t *as input. Output* $b = 1$ *if the authentication tag* t *is valid. Otherwise, output* $b = 0$.

Comparable to SKE, we require the correctness of a MAC scheme:

Definition 12 (MAC correctness)**.** *Let* MAC *denote a message authentication code scheme consisting of the three algorithms as in Definition 11. For every* λ*, every secret key* SK *output by* $\mathsf{MAC\_Gen}$*, and every* $V \in \{0,1\}^*$*, it holds that* $\mathsf{MAC\_Vrfy}(SK, V, \mathsf{MAC\_Tag}(SK, V)) = 1$.

To define the security of a MAC scheme, we again define the threat model and the security goal. In particular, we consider the definition of strongly secure MACs.

Definition 13 (MAC experiment)**.** *Let* MAC *denote a message authentication code scheme consisting of the three algorithms as in Definition 11. For an adversary $\mathcal{A}$ and a security parameter λ, the* MAC *experiment* $\mathsf{Exp}^{MAC-sforge}_{\mathcal{A},\mathsf{MAC}}$ *is defined as:*

1. *The challenger generates a secret key $SK \leftarrow \mathsf{MAC\_Gen}(1^\lambda)$.*
2. *The adversary $\mathcal{A}^{\mathsf{MAC\_Tag}(SK,\cdot)}$ receives input 1^λ, has access to an oracle $\mathsf{MAC\_Tag}(SK, \cdot)$, and outputs (V', t'). Let $\boldsymbol{\mathcal{T}}$ be the set of all (V, t) pairs for which $\mathcal{A}^{\mathsf{MAC\_Tag}(SK,\cdot)}$ used the oracle $\mathsf{MAC\_Tag}(SK, \cdot)$ with value V receiving the authentication tag t as a response.*
3. *The experiment outputs 1 iff $\mathsf{MAC\_Vrfy}(SK, V', t') = 1$ and $(V', t') \notin \boldsymbol{\mathcal{T}}$. $\mathcal{A}^{\mathsf{MAC\_Tag}(SK,\cdot)}$ succeeds if the experiment outputs 1.*

Definition 14 (Strongly secure MAC)**.** *Let* MAC *denote a message authentication code scheme consisting of the three algorithms as in Definition 11 and λ its security parameter.* MAC *is* strongly secure *if for all PPT adversaries $\mathcal{A}$ there exists a negligible function* negl *such that*

$$|\mathsf{Pr}[\mathsf{Exp}^{MAC-sforge}_{\mathcal{A},\mathsf{MAC}} = 1]| \leq \mathsf{negl}(\lambda)$$

2.2.5 Authenticated Encryption (AE)

SKE schemes as presented in Section 2.2.3 protect the confidentiality of a value, and MAC schemes as presented in Section 2.2.4 protect the integrity of values. The goal of an *authenticated encryption* (AE) scheme is to protect the confidentiality and integrity of a value at the same time. An AE scheme can be considered an SKE scheme with the following formal protection guarantees:

Definition 15 (*Authenticated encryption* (AE))**.** *A symmetric-key encryption scheme is an* authenticated encryption *scheme if it is IND-CCA secure and unforgeable.*

IND-CCA security is defined in Definition 10. Unforgeability is defined via the following experiment and corresponding attacker success probability:

Definition 16 (Unforgeable encryption experiment)**.** *Let* SKE *denote a symmetric-key encryption scheme consisting of the three algorithms as in Definition 7. For an adversary $\mathcal{A}$ and a security parameter λ, the* unforgeable encryption experiment $\mathsf{Exp}^{Enc-Forge}_{\mathcal{A},\mathsf{SKE}}$ *is defined as:*

1. *The challenger generates a secret key $SK \leftarrow \mathsf{SKE\_Gen}(1^\lambda)$.*
2. *The adversary $\mathcal{A}^{\mathsf{AE\_Enc}(SK,\cdot)}$ receives input 1^λ, has access to an encryption oracle $\mathsf{AE\_Enc}(SK, \cdot)$, and outputs a ciphertext C. Let $\boldsymbol{\mathcal{T}}$ be the set of all plaintext values for which $\mathcal{A}^{\mathsf{AE\_Enc}(SK,\cdot)}$ used the oracle $\mathsf{AE\_Enc}(SK, \cdot)$.*
3. *Let $V = \mathsf{SKE\_Dec}(SK, C)$. The experiment outputs 1 iff $V \neq \perp$ and $V \notin \boldsymbol{\mathcal{T}}$. $\mathcal{A}^{\mathsf{AE\_Enc}(SK,\cdot)}$ succeeds if the experiment outputs 1.*

Definition 17 (Unforgeable encryption)**.** *Let* SKE *denote a symmetric-key encryption scheme consisting of the three algorithms as in Definition 7 and λ its security parameter.* SKE *is* unforgeable *if for all PPT adversaries $\mathcal{A}$ there exists a negligible function* negl *such that*

$$|\mathsf{Pr}[\mathsf{Exp}^{Enc-Forge}_{\mathcal{A},\mathsf{SKE}} = 1]| \leq \mathsf{negl}(\lambda)$$

Katz *et al.* [48] propose to construct an AE scheme by combining an IND-CPA–secure SKE scheme and a strongly secure MAC scheme, according to the encrypt-then-authenticate approach using two independent keys. Throughout this dissertation, however, we use AES-GCM [49] as AE construction. AES-GCM performs encryption and authentication at the same time using a single key. We use a formal abstraction for this construction and do not go into details:

Definition 18 (AE syntax)**.** *An* authenticated encryption *scheme is a tuple of three PPT algorithms* $\mathsf{AE} = (\mathsf{AE\_Gen}, \mathsf{AE\_Enc}, \mathsf{AE\_Dec})$ *such that:*

$SK \leftarrow \mathsf{AE\_Gen}(1^\lambda)$: *Take a security parameter* λ *as input and output a secret key* SK*.*

$C \leftarrow \mathsf{AE\_Enc}(SK, V)$: *Take a secret key* SK *and a plaintext value* $V \in \{0,1\}^*$ *as input. Calculate an authentication tag* t *for the value* V *and encrypt* V *resulting in a ciphertext* C'*. Output the ciphertext* $C = (C', t)$*.*

$V \leftarrow \mathsf{AE\_Dec}(SK, C)$: *Take a secret key* SK *and a ciphertext* $C = (C', t)$ *as input. Verify the authentication tag* t*. Only if* t *is valid, decrypt the ciphertext* C *and output the plaintext value* V*.*

Note that the IND-CCA security, which is inherent to all AE schemes, guarantees that ciphertexts are different, even if encrypted plaintexts are equal.

2.2.6 Cryptographic Hash Function

The goal of a hash function is to map a finite-length string to a fixed-length string called *hash*. In some use cases, it is desirable that different input strings are mapped to the same hash, e.g., for a hash table. If the input domain is larger than the hash domain, which is the default case, collisions must exist. We, however, are interested in *collision-resistant* hash functions, i.e., it should be infeasible for a PPT adversary to find two strings that have the same hash.

In the following we define *keyed, cryptographic hash functions* and define their collision resistance by stating a threat model and security goal. Afterwards, we define *cryptographic hash functions*, which are a variant of keyed, cryptographic hash functions.

Definition 19 (Keyed, cryptographic hash function syntax)**.** *A* keyed, cryptographic hash function *is a pair of PPT algorithms* $(\mathsf{H\_Gen}, \mathsf{H})$ *such that:*

$k \leftarrow \mathsf{H\_Gen}(1^\lambda)$: *Take a security parameter* λ *as input and output a key* k*.*

$h \leftarrow \mathsf{H}(k, V)$: *Take a key* k *and a value* $V \in \{0,1\}^*$ *as input and output a fixed-size hash* h*.*

Definition 20 (Collision-finding experiment for keyed, cryptographic hash functions)**.** *Let* Π *denote a keyed, cryptographic hash function consisting of the two algorithms as in Definition 19. For an adversary* $\mathcal{A}$ *and a security parameter* λ*, the* collision-finding experiment $\mathsf{Exp}^{Hash-coll}_{\mathcal{A},\Pi}$ *is defined as:*

1. *The challenger generates a key* $k \leftarrow \mathsf{H\_Gen}(1^\lambda)$*.*
2. *The adversary is given* k *and outputs two values* V *and* V'*.*
3. *The experiment outputs 1 iff* $V \neq V'$ *and* $\mathsf{H}(k, V) = \mathsf{H}(k, V')$*. If the experiment outputs 1,* $\mathcal{A}$ *found a collision and succeeds.*

Definition 21 (Collision-resistant, keyed, cryptographic hash function)**.** *Let* Π *denote a keyed, cryptographic hash function consisting of the two algorithms as in Definition 19.* Π *is* collision resistant *if for all PPT adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that*

$$|\mathsf{Pr}[\mathsf{Exp}^{Hash-coll}_{\mathcal{A},\Pi} = 1]| \leq \mathsf{negl}(\lambda)$$

Definition 22 (Cryptographic hash function syntax)**.** *A* cryptographic hash function *is a keyed, cryptographic hash function for which the key is fixed. It consists of one PPT algorithm* H *such that:*

$h \leftarrow \mathsf{H}(V)$: *Take a value* $V \in \{0,1\}^*$ *as input and output a fixed-size hash* h*.*

The collision-finding experiment for cryptographic hash functions is a straightforward modification of the experiment for keyed, cryptographic hash functions. Equally, the definition

for a collision-resistant, cryptographic hash function is a straightforward modification of the definition for a collision-resistant, keyed, cryptographic hash function. Therefore, we do not explicitly state the experiment and definition.

Many hash functions used in practice, e.g., MD5 [50], SHA-256 [51], and SHA-3 [52], have a fixed key and can be considered as implementations of cryptographic hash functions. Throughout this dissertation, we write hash functions but refer to cryptographic hash functions, and we write that a value V is *hashed* meaning that the value is input to the cryptographic hash function evaluation, i.e., $h \leftarrow \mathsf{H}(V)$.

2.2.7 Set Hash Function

Clarke *et al.* [53] propose multiset hash functions, a special kind of cryptographic hash functions. The properties of a multiset hash functions are the following: a multiset, i.e., a finite unordered group in which a value can occur more than once, is mapped to a fixed-size *multiset hash*[1]; there exists an efficient equality check for two multiset hashes; a multiset hash calculated for the multiset $\boldsymbol{M'}$ can be added efficiently to a multiset hash calculated for the set $\boldsymbol{M}$ and the result is equal to a multiset hash calculated for $\boldsymbol{M} \cup \boldsymbol{M'}$; and individual values can be added to a multiset hash incrementally and efficiently.

In this dissertation, we introduce and use *set hash functions*. These hash functions are a variant of multiset hash functions with the following three differences: (1) A set hash function hashes a set to a fixed-size *set hash*[2]. (2) Two set hashes can *only* be added if all values in the underlying sets are distinct. (3) A set hash calculated for the set $\boldsymbol{M'}$ can be subtracted efficiently from a set hash calculated for the set $\boldsymbol{M}$ if $\boldsymbol{M'}$ is a subset of $\boldsymbol{M}$. The result of the subtraction is equal to a set hash calculated for $\boldsymbol{M} \setminus \boldsymbol{M'}$. More formally:

Definition 23 (Set hash function syntax)**.** *A* set hash function *is a tuple of four PPT algorithms* $(\mathsf{SH}, \mathsf{SH\_NI\_Add}, \mathsf{SH\_SS\_Sub}, \mathsf{SH\_Comp})$ *such that:*

$h \leftarrow \mathsf{SH}(\boldsymbol{M})$: *Take a set* $\boldsymbol{M}$ *as input, which contains values from a countable set* $\boldsymbol{B}$*. Output a set hash* h*, which is an element of a set with cardinality* $\approx 2^n$ *for* $n \in \mathbb{N}$*.*

$b \leftarrow \mathsf{SH\_Comp}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M})$ *and* $\mathsf{SH}(\boldsymbol{M'})$ *as input. Output* $b = 1$ *if* $\boldsymbol{M} = \boldsymbol{M'}$ *and* $b = 0$ *otherwise. Note that a set does not define a value order; thus,* $\boldsymbol{M}$ *and* $\boldsymbol{M'}$ *are equal if they contain the same values, independent of the order.*

$h \leftarrow \mathsf{SH\_NI\_Add}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M})$ *and* $\mathsf{SH}(\boldsymbol{M'})$ *as input for which* $\boldsymbol{M} \cap \boldsymbol{M'} = \emptyset$*, efficiently compute* $h = \mathsf{SH}(\boldsymbol{M} \cup \boldsymbol{M'})$*, and output* h*. Note that a set can contain only single value* $\{b\} \in \boldsymbol{B}$*; thus,* $\mathsf{SH\_NI\_Add}$ *can be used to add a single value to an existing set hash.*

$h \leftarrow \mathsf{SH\_SS\_Sub}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M})$ *and* $\mathsf{SH}(\boldsymbol{M'})$ *as input for which* $\boldsymbol{M'} \subseteq \boldsymbol{M}$*, efficiently compute* $h = \mathsf{SH}(\boldsymbol{M} \setminus \boldsymbol{M'})$*, and output* h*. Note that a set can contain only single value* $\{b\} \in \boldsymbol{B}$*; thus,* $\mathsf{SH\_SS\_Sub}$ *can be used to subtract a single value from an existing set hash.*

Clarke *et al.* propose the security notions *set-collision resistance* and *multiset-collision resistance* for multiset hash functions. We slightly modify the *set-collision resistance* to fit to our set hash functions:

Definition 24 (Set-collision resistance)**.** *Let* Π *denote a set hash function consisting of the four algorithms as in Definition 23 and* $\boldsymbol{B}$ *a countable set. A set hash function is* set-collision

1 To clearly differentiate the output of a multiset hash function from a cryptographic hash function, we denote it multiset hash.

2 See argument for multiset hash in Footnote 1.

resistant *if it is computationally infeasible to find a set* $\boldsymbol{M} \subseteq \boldsymbol{B}$ *and a set* $\boldsymbol{M'} \subseteq \boldsymbol{B}$ *such that* $|\boldsymbol{M}|$ *and* $|\boldsymbol{M'}|$ *are polynomial in* $m \in \mathbb{N}$, $\boldsymbol{M} \neq \boldsymbol{M'}$ *and* $\mathsf{SH\_Comp}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'})) = 1$.

The following set hash function construction, called Set-XOR-Hash, is a variant of MSet-XOR-Hash from Clarke *et al.* [53]. The differences follow from the differences between multiset and set hash functions described above. In the Set-XOR-Hash construction, H denotes a keyed, cryptographic hash function.

Definition 25 (Set-XOR-Hash construction)**.** *Set-XOR-Hash is a set hash function construction with the following four PPT algorithms:*

$h \leftarrow \mathsf{SH}^{\mathsf{xor}}(\boldsymbol{M})$: *Take a set* $\boldsymbol{M}$ *as input, which contains values from a countable set* $\boldsymbol{B} = \{0,1\}^n$ *for* $n \in \mathbb{N}$. *Calculate*

$$h = (\mathsf{H}(k, (0 \,||\, r)) \oplus \bigoplus_{m \in \boldsymbol{M}} \mathsf{H}(k, (1 \,||\, m)),\ |\boldsymbol{M}| \bmod 2^n,\ r)$$

where r *is selected uniformly at random from* $\boldsymbol{B}$. *Output* h.

$b \leftarrow \mathsf{SH\_Comp}^{\mathsf{xor}}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M}) = (h, c, r)$ *and* $\mathsf{SH}(\boldsymbol{M'}) = (h', c', r')$ *as input. Output* $b = 1$ *iff*

$$h \oplus \mathsf{H}(k, (0 \,||\, r)) = h' \oplus \mathsf{H}(k, (0 \,||\, r')) \wedge c \equiv c' \bmod 2^n.$$

Otherwise, output $b = 0$.

$h'' \leftarrow \mathsf{SH\_Nl\_Add}^{\mathsf{xor}}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M}) = (h, c, r)$ *and* $\mathsf{SH}(\boldsymbol{M'}) = (h', c', r')$ *as input for which* $\boldsymbol{M} \cap \boldsymbol{M'} = \emptyset$. *Calculate*

$$h'' = (\mathsf{H}(k, (0 \,||\, r'')) \oplus h \oplus \mathsf{H}(k, (0 \,||\, r)) \oplus h' \oplus \mathsf{H}(k, (0 \,||\, r')),\ c + c' \bmod 2^n,\ r'')$$

where r'' *is selected uniformly at random from* $\boldsymbol{B}$. *Output* h''.

$h'' \leftarrow \mathsf{SH\_SS\_Sub}^{\mathsf{xor}}(\mathsf{SH}(\boldsymbol{M}), \mathsf{SH}(\boldsymbol{M'}))$: *Take two set hashes* $\mathsf{SH}(\boldsymbol{M}) = (h, c, r)$ *and* $\mathsf{SH}(\boldsymbol{M'}) = (h', c', r')$ *as input for which* $\boldsymbol{M} \subseteq \boldsymbol{M'}$. *Calculate*

$$h'' = (\mathsf{H}(k, (0 \,||\, r'')) \oplus h \oplus \mathsf{H}(k, (0 \,||\, r)) \oplus h' \oplus \mathsf{H}(k, (0 \,||\, r')),\ c' - c \bmod 2^n,\ r'')$$

where r'' *is selected uniformly at random from* $\boldsymbol{B}$. *Output* h''.

Clarke *et al.* [53] prove that MSet-XOR-Hash is a set-collision resistant multiset hash function. The security proof is a reduction to the hardness of breaking the underlying pseudorandom function. The security of Set-XOR-Hash follows from this proof and we refer the interested reader to the paper.

2.2.8 Merkle Tree

Merkle trees are mainly used for efficient and secure integrity checks for stored values. A Merkle tree is a binary tree computed over n values $\boldsymbol{V} = (V_0, \ldots, V_{n-1})$. W.l.o.g. we assume that n is a power of 2 in the following description. Each leaf node X_i of the tree $\boldsymbol{T}$ stores one value V_i. Each other node stores a hash of its two children, i.e., if a node X_i has the child nodes X_u and X_w, which store U and W, then X_i stores the hash $h_i = \mathsf{H}(U||W)$ with H being a hash function and $||$ a concatenation of values. The root node $T.root$ of the tree stores the so-called *root hash*. Figure 2.1 shows a Merkle tree example.

A typical use case for Merkle trees is that a trusted party (e.g., a client or enclave) wants to store values at an untrusted party (e.g., a cloud server). To protect the integrity of the outsourced data, the trusted party calculates a Merkle tree over the stored values; outsources

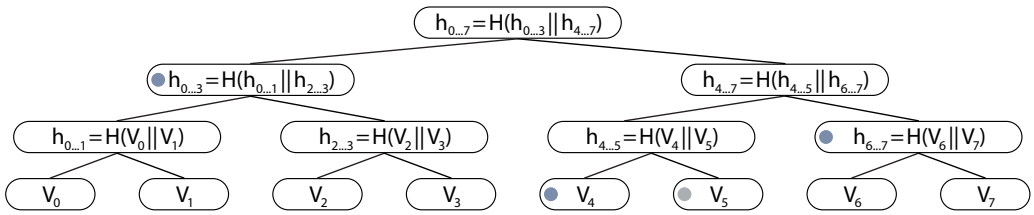

Figure 2.1: Merkle tree example. H is a hash function and || a concatenation of values. The integrity verification of the value marked with a gray dot requires the content of the nodes marked with a blue dot.

the data and the Merkle tree; and only retains the root hash. If the trusted party requests a value V_i and wants to check the integrity of the returned value, it additionally needs the content of the Merkle tree nodes adjacent to the path from X_i to $T.root$. With this content, the trusted party can recalculate all hashes up to the root hash. The integrity of V_i is proven if the calculated root hash is equal to the stored root hash. As the path from X_i to $T.root$ is logarithmic in the number of stored values, the trusted party needs content from a logarithmic number of tree nodes.

For instance, assuming that the trusted party wants to perform an integrity check for the value V_5 in our example presented in Figure 2.1 (marked with a gray dot). Then, the trusted party only needs V_4, $h_{6...7}$, and $h_{0...3}$ (marked with a blue dot) to recalculate the root hash.

The security of a Merkle tree is defined using its collision resistance, which follows from the collision resistance of the used hash function. More formally, let MT be the function that takes n input values $\boldsymbol{V} = (V_0, \ldots, V_{n-1})$, computes a Merkle tree using a collision resistant hash function H, and outputs the root hash. Then, MT is collision resistant for any fixed n.

3 Trusted Execution Environments (TEEs)

Traditional isolation mechanisms protect user applications from each other using memory control, protect privileged code from user applications using multiple privilege levels, and protect operation systems from each other using virtualization. However, they do not protect user applications and the data processed by these applications from privileged code, e.g., the OS, hypervisor, or firmware. Additionally, they do not protect against physical attacks, e.g., bus tapping or cold boot attacks [54]. TEEs can protect (parts of) user applications against privileged code and physical attacks. This protection guarantees make TEEs especially valuable in a cloud computing scenario as TEEs allow data owners to securely execute applications at an untrusted cloud provider.

The term "trusted execution environment" was coined by OMTP Limited [55] in 2009. TEE architecture specifications [56], [57] and papers defining TEEs [58], [59] followed, but the community has not yet converged on a commonly accepted definition. A generic definition is particularly hard to achieve, because the features of approaches calling themselves TEEs differ in many regards [23], [60]–[64]. In this dissertation, we define a TEE as follows:

Definition 26 (*Trusted execution environment* (TEE))**.** *A* trusted execution environment *provides an isolated processing environment for user-defined code and data, i.e., the confidentiality and integrity of code and data processed in this environment are protected against other software and physical accesses.*

In addition, we define the terms *enclave* and *enclave memory* as follows:

Definition 27 (Enclave)**.** *An* enclave *is a monolithic software entity consisting of user-defined code that processes data in an isolated processing environment provided by a TEE.*

Definition 28 (Enclave memory)**.** *The* enclave memory *encompasses all memory regions that are exclusively accessible by a corresponding enclave due to the protection provided by a TEE.*

Furthermore, we require that a TEE supports the capabilities listed in Table 3.1.

An enclave is shipped as a binary to the cloud provider, and we could define the binary's size in bytes as the *enclave size*. However, the developer, who develops the enclave, and the data owner, who might want to inspect the enclave's source code, do not use the binary file for these tasks. Instead, they work with the enclave code in a programming language, which is often determined by a *software development kit* (SDK) accompanying the TEE. Additionally, a legacy application, for which parts should be protected by an enclave, might determine the programming language. Thus, we define the enclave size as follows:

Definition 29 (Enclave size)**.** *The* enclave size *is the size of an enclave's source code in LOC.*

In Section 3.1, we present details of the only commercially available TEE achieving the capabilities listed in Table 3.1[1]—Intel SGX. Throughout this dissertation, we assume that Intel SGX is used as a TEE. However, Intel SGX can be replaced by any other TEE that provides

1 During the timeframe in which the research for this dissertation was performed.

Cap.	Description
C1	Execution of user-defined enclaves.
C2	Confidentiality protection for data, i.e., an attacker[a] cannot observe any information about the processed data.
C3	Integrity protection for code and data, i.e., an attacker[b] cannot affect the executed code and the integrity and freshness[c] of the processed data. The attacker might deny execution, but if the program executes, it returns the correct output.
C4	Remote attestation mechanism, which enables the data owner to cryptographically verify that a specific enclave has been loaded into an isolated processing environment of a specific TEE.
C5	Remote provisioning mechanism, which enables the data owner to provide sensitive data to the enclave without leaking the data to the untrusted cloud provider or other parties.
C6	Sealing mechanism, which stores data in untrusted storage while protecting the confidentiality, integrity, and freshness of the data.
C7	Source of randomness, which is trusted and accessible by the enclave.

[a] The attacker might be other software, e.g., user applications, enclaves, the OS, or device firmware; or entities, e.g., the cloud provider, hackers, or governments.
[b] See Footnote a.
[c] Freshness guarantees that data is returned in the latest version. In other words, a freshness guarantee protects against replay attacks.

Table 3.1: TEE capabilities required by this dissertation.

the required capabilities. In Section 3.2, we describe known attacks on Intel SGX, which threaten the fulfillment of the listed capabilities. For all attacks, we also depict mitigations. In Section 3.3, we differentiate related technologies, which we do not consider a TEE, and we present to which extent other commercially available TEEs fulfill the capabilities listed in Table 3.1.

3.1 Intel Software Guard Extensions (Intel SGX)

Intel SGX is an instruction set extension that is available in Intel Core processors since the Skylake generation and in Intel Xeon processors since the Kaby Lake generation, making Intel SGX a widely available TEE. Its main goal is to provide isolated processing for enclaves. In other words, Intel SGX guarantees the integrity of enclave code, and it guarantees confidentiality and integrity for data processed by the enclave, even in an untrusted environment.

In the following subsections, we introduce details of Intel SGX[2] as far as necessary to understand this dissertation. In sections 3.1.1, 3.1.2, 3.1.3, and 3.1.4, we explain how Intel SGX achieves the capabilities listed in Table 3.1 using memory isolation, application separation, attestation, and data sealing, respectively. Then, we describe the *trusted computing base* (TCB) of Intel SGX–enabled applications in Section 3.1.5. In Section 3.1.6, and 3.1.7, we introduce auxiliary capabilities of Intel SGX, which we use in this dissertation. More details about Intel SGX can be found in the literature [20]–[28].

3.1.1 Memory Isolation

On system startup, Intel SGX hardware reserves a dedicated portion of the system's RAM. This portion is called *processor reserved memory* (PRM) and can be configured up to a maximum of 128 MB. While PRM data is cache resident, the CPU performs access control checks denying all non-enclave access, including access by privileged software. Before moving cached PRM data

2 Precisely, we describe Intel SGXv2, the most current version at the time of writing.

to the RAM, an on-chip memory encryption engine performs encryption at the granularity of cache lines. Before PRM data is loaded from the RAM into CPU caches, this engine decrypts the data, checks the data's integrity, and verifies the data's freshness. Consequently, not even an attacker with physical access to the RAM can retrieve plaintext data, modify data, or rollback data in the PRM (without being detected).

The PRM contains a region called the *enclave page cache* (EPC) comprising memory pages of 4 kB each for enclave code and data. For a PRM with 128 MB, the EPC has about 96 MB available for enclaves and the EPC is shared between all enclaves running on the system. The remaining space of the PRM is used for management capabilities. For instance, a so-called *enclave page cache map* (EPCM) stores one entry for each memory page in the EPC. Each EPCM entry stores a validity bit, the page type, the corresponding enclave, the virtual address, and permission bits. On each enclave page access, Intel SGX uses the EPCM for a trusted access control check. In particular, enclaves are prevented from accessing pages of other enclaves.

The combination of access control checks for cache-resident data, memory encryption with its integrity and freshness checks, and in-enclave page access checks achieve capabilities C2 and C3 (from Table 3.1).

To mitigate the restriction of 96 MB EPC memory, the OS can swap out EPC pages to the remainder of the system's RAM or to disk. This process is similar to paging used in most OSes [65]. The difference is that Intel SGX ensures integrity, confidentiality, and freshness when pages are swapped in and out. Intel SGX's EPC paging mechanism increases the usable enclave memory, but introduces a severe performance overhead [66], [67].

Consequently, in the case of Intel SGX, the enclave memory (see Definition 28) can encompass parts of the EPC, the entire EPC, and/or swapped out EPC pages.

3.1.2 Application Separation

To execute user-defined applications using Intel SGX, the developer has to divide the code into two parts: an *untrusted part* and an isolated, trusted part—the enclave (cf. C1). All communications between the two parts use an interface specified during design time. The interface can offer *enclave calls* (ECalls) invocable by the untrusted part and *outside calls* (OCalls) invocable by the enclave. Figure 3.1 illustrates the application separation and the calls between the two parts.

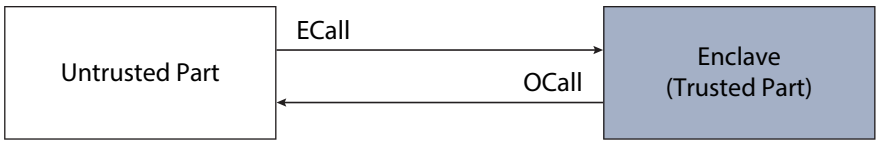

Figure 3.1: Application separation.

Whenever the untrusted part invokes an ECall, the CPU switches to a special enclave mode (similar to a switch from user mode to kernel mode in most OSes), suspends debugging features, backs up the current processor state, potentially copies parameters to the enclave memory, and transfers control to a defined entry point in the enclave. The enclave code is executed until it ends explicitly by a return or implicitly by an interrupt or an exception. On explicit exits, the CPU might copy data out of the enclave memory to the unprotected memory, scrubs the processor state, restores the old processor state, and exits the enclave mode. On implicit exits, the CPU stores the current processor state to an EPC page before scrubbing the state. If the enclave is resumed, the CPU loads the stored state. The switches into and out of the enclave are called *context switches*, and they introduce a non-negligible overhead ($>$ 8000 cycles [68]) due to the described operations.

The enclave can invoke OCalls to perform operations it cannot do on its own, e.g., system calls, I/O operations, and other OS operations. These operations are untrusted, because the OS is untrusted, and the enclave has to take care of data protection. The context switch and the induced overhead are similar to the invocation of ECalls.

The untrusted part is executed as an ordinary process within the virtual memory address space, and it is responsible for setting up the enclave. The enclave memory is mapped into the virtual memory of the untrusted part's process. This mapping can be used as a further mitigation for the restricted 96 MB EPC memory, because the enclave is allowed to access the entire virtual memory of its host process, which can contain the whole system's RAM and disk space. Inherently, Intel SGX does not guarantee integrity or confidentiality for data outside the PRM, but carefully written enclaves can provide this guarantee. Intel SGX denies all accesses from the untrusted part to the enclave memory.

Furthermore, the enclave has access to the RDRAND instruction, which generates true random numbers from hardware (cf. C7).

3.1.3 Attestation

Intel SGX supports two types of attestation: local and remote attestation (cf. C4). Both are based on the enclave's identity, defined as its *measurement*, which is calculated as follows: While the CPU loads the initial code and data of an enclave into the enclave memory, it computes the hash of all pages (called *measuring* in Intel SGX terminology). Once the enclave is initialized, the hash is finalized, becomes the enclave's measurement, and cannot be changed anymore.

Local attestation is used by an attested enclave to prove to a target enclave that it has a specific measurement and that they are hosted by the same Intel SGX–enabled CPU. On a high level, the local attestation is performed as following:

1. The target enclave sends its measurement to the attested enclave.
2. The attested enclave passes the target enclave's measurement to Intel SGX's EREPORT instruction. First, EREPORT fills a so-called *report* with the attested enclave's measurement, further attributes of the attested enclave, and optionally with *report data*. Then, EREPORT uses the target enclave's measurement and a secret embedded in the processor to derive a symmetric *report key*. With this key, EREPORT calculates a MAC tag over the report. Finally, EREPORT returns the report and MAC tag to the attested enclave, which forwards both to the target enclave.
3. The target enclave calls another Intel SGX instruction, which derives the report key using the target enclave's measurement and a secret embedded in the processor. Using this key and the received MAC tag, the target enclave can verify the received report.

Remote attestation is used by an attested enclave to prove to a remote party that it has a specific measurement and that it is hosted on an Intel SGX–enabled CPU. On a high level, the remote attestation is done as following:

1. The remote party issues a challenge, e.g., a nonce, to the attested enclave.
2. The attested enclave performs local attestation with the *quoting enclave* (QE), which is an *architectural enclave* provided by Intel[3]. During this local attestation, the challenge is used as report data.

3 Some Intel SGX functionality is implemented as an enclave, because it is too complex for a hardware realization. These enclaves are called architectural enclaves and besides the *quoting enclave* (QE), there are two other architectural enclaves: First, the *launch enclave* (LE) deciding which other enclaves can be started. Second, the *provisioning enclave* (PE), which uses a key fused into the CPU to proof the CPU's authenticity to an Intel service. After successful verification, the Intel service sends an *attestation key* to the PE. The PE encrypts this key and stores it in the untrusted environment. Besides the PE, only the QE can access the attestation key.

3. After validating the report, the QE replaces the MAC with a signature using an *attestation key*[4]. The resulting signed report is called *quote* and it is sent to the remote party.
4. The remote party uses Intel's attestation service to verify the quote's signature and compares the challenge contained in the quote to the challenge send in the first step. If both checks are successful, the remote party can compare the measurement contained in the quote to an expected value.

The remote attestation feature also allows to establish a secure channel between an external party and an enclave, e.g., using a Diffie-Hellman key exchange. This secure channel can be used to deploy sensitive data, e.g., credentials, cryptographic keys, or access permissions, into the enclave (cf. C5).

3.1.4 Data Sealing

Inherently, Intel SGX enclaves are stateless, i.e., all state information is lost when the enclave is terminated, unless the state is explicitly persisted. To preserve data across multiple enclave runs, Intel SGX offers data sealing (cf. C6). This process uses a *sealing key* to encrypt and integrity-protect data. Afterwards, the data can be stored outside of the enclave in untrusted memory, and only an enclave with the same sealing key can unseal the data.

Intel SGX offers two policies for data sealing:

- The sealing key can be derived from the enclave measurement, which yields a different key for any change impacting the measurement.
- The sealing key can be derived from the enclave developer's public key. This allows an enclave developer to migrate sealed data to an updated enclave[5] or to share sealed data between multiple enclaves[6].

To provide a concise description, we assume that the sealing key is derived from the enclave measurement throughout this dissertation.

3.1.5 Trusted Computing Base (TCB)

According to Lampson *et al.* [69], the *trusted computing base* (TCB) can be defined as "a small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security." In a secure, outsourced data processing environment, this translates to all software and hardware having access to plaintext data. As bugs or vulnerabilities in the TCB can compromise the security of the whole system, the TCB should be small.

The TCB of an application executed with Intel SGX encompasses the user-defined enclave, architectural enclaves provided by Intel (i.e., the quoting enclave, launch enclave, and provisioning enclave), and Intel SGX related parts of the CPU[7]. The developer can only influence the TCB by keeping the enclave size small.

4 See Footnote 3.
5 Each enclave has a *security version number* (SVN), which should be incremented on security relevant enclave changes. Additionally, each CPU has an SVN, which is incremented by Intel on security relevant changes to the CPU. A migration of sealed data is only possible from enclaves with lower SVNs to enclaves with higher SVNs or from CPUs with lower SVNs to CPUs with higher SVNs.
6 Sharing sealed data between enclaves is only possible if both enclaves have the same product ID.
7 We consider the CPU's microcode a part of the CPU.

3.1.6 Protected File System Library

The protected file system library is shipped with Intel SGX's SDK and provides a subset of the regular C file API, e.g., file creation, file writing, and file reading. On write operations, the library separates data into 4 kB chunks, ensures the data's integrity with a Merkle hash tree (see Section 2.2.8) variant, and encrypts each chunk with AES-GCM before storing the chunk in untrusted memory. When file chunks are loaded back into the enclave, the library verifies confidentiality and integrity. A developer can provide the encryption key manually, or the library can derive the key automatically from the sealing key. At any point, only one file handle can be open for writing, but many handles for reading.

3.1.7 Switchless Calls

A primary performance overhead of applications using Intel SGX are switches into and out of an enclave. Intel SGX's SDK supports switchless calls, a technique to reduce this overhead. Calls into the enclave are replaced by writing tasks into an untrusted buffer and enclave worker threads asynchronously perform the tasks. Calls out of the enclave are written into a separate untrusted buffer and untrusted threads perform the tasks. As a result, costly context switches are reduced, which severely decreases the performance overhead.

3.2 Attacks on Intel SGX and Mitigations

In this section, we discuss attacks on Intel SGX differentiated into four categories: inherent side-channel attacks, i.e., attacks that result from deliberate design decisions; enclave code exploits, i.e., attacks that result from exploitable enclave code; enclave-based malware, i.e., attacks that result from the fact that Intel SGX protects executed (malware) code; and processor bugs i.e., attacks that result from bugs in Intel processors enabling the circumvention of Intel SGX's protection guarantees. For all categories, we also discuss mitigation strategies.

3.2.1 Inherent Side-Channel Attacks

A side channel is any resource that is influenced by the execution of an application and can be observed by an attacker. If an attacker uses a side channel to gain sensitive information, we call it a side-channel attack. Such attacks have been known long before Intel SGX was presented [70] and a number of variants have been studied, e.g., software-timing [71], cache-timing [72]–[74], and power consumption [75] side-channel attacks. From the start, Intel communicated that Intel SGX is not designed to handle side-channel attacks and that it is the developer's responsibility to address these attacks [24], [25]. Thus, not surprisingly, many papers show that Intel SGX is susceptible to side-channel attacks.

All attacks described in the following assume that the attacker knows the source code of the enclave. This is a valid assumption as Intel SGX inherently does not protect the confidentiality of the code[8]. The attacks, however, are only feasible if the enclave has secret-dependent memory access patterns. Table 3.2 provides a brief overview about known side channels of Intel SGX. Note that we do not consider hardware-based side-channel attacks.

The page-fault side channel is inherent to Intel SGX for the following reason: The OS is untrusted, yet manages the enclave's pages. Thus, the OS can induce page faults by restricting page access and the page faults can be used to generate a precise trace of the enclave's

8 An Intel SGX extension denoted Intel SGX Protected Code Loader can be used to protect the confidentiality of an enclave. At build time, the extension encrypts the enclave's binary and at enclave load time, the extension decrypts the binary. See Intel SGX's developer reference for more information [24].

Side channel	Weakness	Exploitation method	Goal
Page-fault	OS is responsible for enclave's page management	Inject page faults	Page access trace
Cache-timing	Enclaves use same L1 & L2 cache as other applications on the same core	Overwrite cache lines + time measurement	Cache access trace
Branch-prediction	Enclaves use same branch history as other applications on the same core	Branch shadowing + last branch record read	Control flow

Table 3.2: Overview of Intel SGX's side channels.

page accesses. Research papers show how to combine this attack with code knowledge to extract sensitive information from cryptographic functions [76] and enclaves processing end-user functions, e.g., font rendering, spell-checking, and image processing [32].

Furthermore, enclaves use the same L1 & L2 cache as other applications on the same core. This fact can be used for a cache-timing side-channel attack: A malicious application running on the same CPU core as a victim enclave can first evict (all) cache entries. After cache accesses by the enclave, the malicious application can observe which cache lines are evicted by measuring access times to all cache lines. Combined with code knowledge, papers use this side channel to extract cryptographic keys from an enclave [33], [77], [78] and to identify DNA sequences processed by a genome indexing algorithm in an enclave [77].

Even Intel's implementation for remote attestation, which applies several programming techniques to protect against side-channel attacks, was vulnerable to cache-timing [79]. The private remote attestation key (of a quoting enclave in debug mode) could be extracted only because the implementation leaks the number of loop iterations during signature creation.

For performance reasons, modern CPUs avoid pipeline stalls by predicting the outcome and target of branches. To improve predictions, the CPUs store a branch history. Intel SGX does not clear the branch history when leaving the enclave mode, which can be used for a branch-prediction side-channel attack: The attacker writes a "shadow code", which has the same branch instructions as the victim enclave and a collision on specific branch target address parts. Throughout the enclave processing, the attacker interrupts the enclave with a high frequency, runs the shadow code on the same core as the enclave, and reads the last branch record, which logs if a branch prediction was correct. With this information, the attacker can infer the control flow of the enclave. Lee *et al.* [34] use this side channel to extract cryptographic keys.

Mitigations. Different mitigations are possible against the page-fault side channel, e.g., (automatic) enclave code transformation making page accesses independent of enclave secrets [76], [80], in-enclave page management preventing page trace leakage to the OS [76], or code randomization for enclaves [81]. Other papers propose detection of page-fault attacks from within the enclave by measuring frequent interruptions of the enclave execution [82], [83]. Possible mitigations against the cache-timing side channel are the following: cache partitioning between trusted and untrusted processes [77], hardware transactional memory preventing the observation of cache misses on sensitive code [84], and compiler-based data location randomization [85]. To mitigate the branch-timing side channel, Intel SGX could flush the branch history when leaving enclave mode [34], the CPU could use a separate branch history for each enclave, or the developer could use an obfuscation compiler and a runtime randomizer for control flow randomization [86].

Potentially, Intel could implement improvements to protect against some side-channel attacks in future versions of Intel SGX. For instance, handling the page table inside the enclave could protect against the page-fault side channel; having a strict cache separation between enclave and non-enclave data could protect against the cache-timing side channel; and clearing the branch history on leaving the enclave mode could protect against the branch-prediction side channel. Still, it is mainly the enclave developer's task to protect against side-channel leakage, e.g., by using the presented mitigations. Preventing all side channels is hard if the enclave size is large or if the enclave contains (legacy) code that was not designed for an enclave. Therefore, the enclave size should be kept small and an enclave should be designed specifically for a given problem.

3.2.2 Enclave Code Exploits

Besides side-channel attacks, enclaves can have vulnerabilities in their code, which are directly exploitable by an attacker.

Lee *et al.* [35] present Dark-ROP, an exploitation technique that uses memory-corruption vulnerabilities in an enclave for *return-oriented programming* (ROP) [87]. Dark-ROP uses fuzzing on the enclave's ECalls and Intel SGX's exception handling to detect buffer overflow vulnerabilities. Then, Dark-ROP searches ROP gadgets[9] in the enclave without requiring access to the plaintext source code. If it finds the necessary gadgets, Dark-ROP can read/inject data from/into the enclave memory and force the enclave to perform protected functionality. For instance, an attacker can use Dark-ROP to extract arbitrary enclave data and to break the secure communication channel between remote party and enclave.

Weichbrodt *et al.* [36] present AsyncShock, a tool to exploit synchronization bugs in multi-threaded enclaves. The attacker first searches for synchronization bugs in the enclave's source code and then defines a "playbook" to exploit the bug. The playbook is a list of events, e.g., thread creation, segmentation fault, and timer expiration, and corresponding actions, e.g., pausing a thread, starting a timer, or changing page permissions. As the untrusted OS controls the page table, the attack becomes more reliable than on non-enclaved applications. The authors use AsyncShock to bypass access control checks inside an enclave and to modify an enclave's control flow.

Mitigations. To mitigate Dark-ROP, the developer can, e.g., (automatically) eliminate the necessary gadgets, integrate control flow integrity, and use address space randomization [35]. Possible mitigations against AsyncShock are sanitization of user input and prohibition of threading [36]. Besides these mitigation strategies, the developer is responsible for writing secure enclave code, a task which is facilitated by a small enclave size. Additionally, formal software verification [88] can be used if the enclave size is small.

3.2.3 Enclave-based Malware

Intel SGX protects enclaves from other applications, including malware, but it does not protect non-enclave software from malware inside an enclave. Before Intel SGX was available on the market, Rutkowska [89] outlined that an enclave host cannot know what the enclave does. The enclave might be a generic code loader that fetches encrypted malware from an external source, decrypts the malware, and executes it. Intel SGX's strong confidentiality and integrity guarantees hide the executed malware from anti-virus software in ring 3 and even monitoring software running in ring 0. Davenport [90] describes an enclave which is part of a botnet. The

9 ROP gadgets are (short) instruction sequences present in existing code and ending with a return instruction.

command and control server performs remote attestation with the enclave and then uses the enclave to execute arbitrary payload.

Schwarz *et al.* [37] present the first implementation of an attack using the remote loading of malware. They use this technique to conceal a cache side-channel attack on co-located enclaves. With this hidden cache side-channel attack, they can reconstruct the private key used in a co-located RSA calculation.

In another paper, Schwarz *et al.* [38] describe how malware in an enclave can overcome the limitation that it cannot issue system calls. They explain how a malicious enclave can find read and write gadgets in its corresponding untrusted part. This gadget search is possible without cooperation of the untrusted part, without the necessity of bugs in the untrusted part, and without being detectable by the OS. The enclave uses the gadgets for ROP on its untrusted part allowing the enclave to invoke arbitrary system calls.

Mitigations. Intel uses the following strategy to protect against enclave-based malware: Only enclaves in release mode[10] provide the guarantees described in Section 3.1. An architectural enclave provided by Intel, the launch enclave, prevents release mode enclaves from starting if the used public key is not on a whitelist managed by Intel. To get a public key whitelisted, enclave developers have to meet defined development and security standards and enter a commercial use license with Intel [91]. On the one hand, the fear of losing the possibility to create release mode enclaves might prevent an enclave developer from incorporating malware into an enclave. On the other hand, the enclave developer is the only party to gain from enclave-based malware and the presented strategy does not enforce malware protection. Mainly, it is in the enclave host's interest to prevent enclave-based malware. A reasonable mitigation strategy to uncover malware in an enclave is a manual inspection of the enclave code. The enclave size should be small to facilitate this inspection.

3.2.4 Processor Bugs

The attacks on Intel SGX presented so far are out-of-scope of Intel SGX's threat model, i.e., Intel does not guarantee protection against these attacks. It is the developer's responsibility to carefully write enclaves that are resistant to side-channel attacks and enclave code exploits. Furthermore, it is the enclave host's responsibility to prevent enclave-based malware. In this section, we describe attacks that should not be possible under Intel SGX's threat model, but exploitable bugs in Intel processors circumvent the protection guarantees. The attacks do not require enclaves with secret-dependent memory access patterns to be successful. We provide the name of the attacks and briefly describe them, because they were part of many Intel SGX related technology news over the last years [92]–[95].

Spectre- and Meltdown-type attacks. Modern CPUs massively parallel-process instructions with multiple execution units. To keep the pipeline full at all times, the CPUs use speculative execution, i.e., they predict the outcome of (conditional) branches and data dependencies to process instructions before the decision is evaluated. Furthermore, they use out-of-order execution: Instructions are processed as soon as the required execution unit and source operands become available, even if the incoming order is different. Intermediate results are buffered in the microarchitectural state (e.g., CPU caches and the line file buffer) and committed to the architectural state (e.g., registers and main memory) according to the incoming order.

However, after instructions are processed speculatively, the predictions might turn out wrong, and after instructions are processed out-of-order, an exception might occur. In both cases, the intermediate results are discarded and do not reach the architectural state. The instructions

10 Besides release mode, Intel SGX offers a simulation, a debug, and a pre-release mode for various stages of enclave development. These modes do not protect the enclave code and the processed data.

that are first executed speculatively or out-of-order and then discarded, are called *transient instructions*. The CPU discards transient instructions, but they may still leave traces in the microarchitectural state. The renowned Spectre [96] and Meltdown [97] attacks on Intel CPUs use these effects as follows: they prepare the microarchitectural state; send a trigger instruction, which eventually will be recognized as a misprediction/exception; abuse a sequence of transient instructions as the sending end of a covert channel; and recover the information from the covert channel after a misprediction/exception is recognized by the CPU. *Spectre-type* attacks exploit transient instructions due to mispredictions, and *Meltdown-type* attacks exploit transient instructions due to exceptions. Both attack types subvert software- and hardware-based memory isolation boundaries between, e.g., different user spaces, user and kernel space, and different *virtual machines* (VMs). In the following, we only describe attacks that are (also) applicable to Intel SGX.

SGXPectre [39] is a Spectre-type attack. It prepares an array in untrusted memory, analyzes the victim enclave's code, and uses non-enclave code for a targeted modification of the CPU's branch prediction. As a result, the processed enclave accesses the prepared array. The accessed array index corresponds to a value stored at an attacker-defined enclave memory address. Finally, SGXPectre loads each array entry and determines the secret by comparing the loading times. This works, as only one array entry is cached if cache noise can be suppressed. The authors demonstrate how SGXPectre can be used to extract register values of enclaves and to extract cryptographic keys from Intel SGX's architectural enclaves. With these keys, an attacker can launch arbitrary enclaves in production mode, forge local and remote attestation responses, and decrypt sealed data. To be successful, SGXPectre requires knowledge of the victim enclave's source code and vulnerabilities within the enclave.

Foreshadow [40] is a Meltdown-type attack. It first allocates a buffer in untrusted memory with 256 slots and provokes the enclave to load a secret into the CPU's L1 cache. Then, it dereferences the enclave secret, which will eventually lead to a page fault. However, in the meantime, transient instructions load one of the 256 slots corresponding to the secret. After the page fault is retired, the attack uses cache timing to determine the accessed slot. Byte by byte, the attacker can leak arbitrary data from the enclave. The authors demonstrate that Foreshadow can extract cryptographic keys from Intel SGX's architectural enclaves. Foreshadow relies solely on the elementary behavior of Intel CPUs and does not require side-channel vulnerabilities, vulnerable code within the enclave, or the victim enclave's code.

ZombieLoad [41] is a Meltdown-type attack. It uses the fact that faulting load instructions on an internal CPU buffer may transiently read stale values belonging to previous memory operations, before being reissued. The attack can recover these stale values using known covert channels, but the attacker cannot define the leaked enclave memory address. Instead, ZombieLoad leaks values currently loaded or stored by the CPU core. The authors show that ZombieLoad can be used to leak an enclave's sealing key.

CacheOut [98] is a Meltdown-type attack similar to ZombieLoad. However, it can bypass Intel's countermeasures against ZombieLoad, the attacker can select the enclave memory that should be leaked, and it is effective even without hyperthreading. CacheOut uses the observation that data evicted from the L1 cache occasionally ends up in a CPU internal buffer, from where the data can be leaked with techniques comparable to ZombieLoad. The authors use CacheOut to dump enclave memory, even if the enclave is idle. In a subsequent work, called SGAxe [99], the authors describe more details on CacheOut attacks on Intel SGX and use these attacks to extract an enclave's sealing key and the CPU's attestation key.

Voltage attack. The Pludervolt [100] attack uses a privileged software interface to regulate the processor's voltage. By undervolting the CPU, Plundervolt injects faults in the enclave computation, which corrupts the integrity guarantees for functionally correct Intel SGX enclaves.

The authors demonstrate that these faults can break, e.g., the integrity of Intel SGX's key derivation, local attestation report MAC tags, and AES-NI calculations. The authors only state that such attacks could break Intel SGX's security, but leave the exploitation to future work. Additionally, the authors show that Pludervolt can cause memory safety problems as incorrect index calculations can result in out-of-bounds array accesses and wrong allocation size computations to subsequent heap corruption or disclosure. The authors demonstrate that these memory safety problems might write enclave secrets to untrusted memory.

Mitigations. For all processor bugs presented in this section, Intel provided a microcode update to resolve[11] the problem—before or shortly after the problem was made public. With each security relevant microcode update Intel increases the CPUs' SVN, which Intel SGX uses in all key derivation requests and measurement calculations. The CPU's SVNs is also embedded in each local attestation report and thus part of each quote used for remote attestation. Consequently, data encrypted by enclaves running with a new microcode is protected from enclaves with an older microcode, and a remote party can verify the used microcode. In new CPU generation, some problems are also solved via architectural changes.

3.3 Other TEEs and Related Technologies

In this chapter, we first differentiate related technologies that we do not consider a TEE. Afterwards, we describe commercially available TEEs and present to which extent they achieve the capabilities listed in Table 3.1. We do not consider TEEs limited to academic research, e.g., Sanctum [62], Bastion [63], and AEGIS [64], because they only propose processor architectures. It would require processor manufacturing capabilities to use these TEEs.

3.3.1 Related Technologies

HSM. *Hardware security modules* (HSMs) [101]–[103] are dedicated, tamper-resistant extension hardware. They are mainly used to protect sensitive secrets, e.g., passwords, hashes, and private keys; and to execute cryptographic functions, e.g., encryption, decryption, and signing. In theory, they can host arbitrary functionality, but they are costly and have limited processing power. As this is unsuitable for complex user-defined code, we do not consider them as TEEs.

TPM. *Trusted platform modules* (TPMs) [104] are dedicated, tamper-resistant coprocessors, which are typically located on the motherboard. They can be used for remote attestation, data sealing, and cryptographic operations, e.g., random number generation, hash functions, and encryptions. For these operations, TPMs contain a small, dedicated memory, which protects the confidentiality and integrity of the processed data against software and physical attacks. However, TPMs do not support user-defined code; thus, we do not consider them as TEEs.

FPGA. *Field programmable gate arrays* (FPGAs) [105] are hardware devices containing blocks of logic and interconnects between these blocks, which are configurable after manufacturing. To protect the integrity of sensitive code, a developer can write a key into a write-only memory of the FPGA. Afterwards, the developer can sign and encrypt a binary, which the FPGA can decrypt and authenticate during boot. This process, however, does not scale as the developer has to manage FPGAs, i.e., the developer has to order FPGAs, write keys into each FPGA, and ship the FPGAs to cloud providers. Therefore, we do not consider FPGAs as TEEs.

11 Intel's microcode update against Foreshadow can only prevent the attack if hyperthreading is disabled.

3.3.2 Commercially Available TEEs

Now, we describe three commercially available TEEs: ARM TrustZone, *AMD SEV-Secure Nested Paging* (AMD SEV-SNP), and *IBM Secure Execution* (IBM SE). Table 3.3 shows a comparison between Intel SGX and these TEEs based on the capabilities listed in Table 3.1.

	Intel SGX	ARM TrustZone	AMD SEV-SNP	IBM SE
C1 (user-defined enclaves)	●	●	●	●
C2 (confidentiality protection)	●	◐	●	●
C3 (integrity protection)	●	◐	●	●
C4 (remote attestation)	●	○	●	–
C5 (remote provisioning)	●	○	●	–
C6 (sealing)	●	○	●	●
C7 (source of randomness)	●	●	●	–
Hardware TCB	CPU chip package	CPU chip package	CPU chip package	CPU chip package
Software TCB	User enclave, architectural enclaves	Secure world (firmware, OS, application)	Entire VM	Entire VM

Table 3.3: Comparison of TEEs. See Table 3.1 for more detailed description of capabilities C1–C7. The symbols represent that a capability is supported (●), partially supported (◐), not supported (○), or that the support is unknown (–).

ARM TrustZone. ARM TrustZone [61], [106] is a set of security extensions for processors and microcontrollers based on the ARM architecture. In the following, we present TrustZone's capabilities as described by ARM. However, as ARM is an intellectual property provider and not a chip manufacturer, the manufacturer is free to add or remove capabilities.

TrustZone partitions the system's resources between a "normal world" and "secure world". Software in the secure world can compromise any level in the normal world's software stack, but software in the normal world can only access the secure world at well-defined locations. A special bit in memory addresses signals whether a memory access belongs to the normal or the secure world, and the CPU sets the bit to zero for normal world address translations. CPU caches use this bit in all addresses, effectively separating the cache entries for the two worlds. Other hardware modules, e.g., RAM and DMA controllers, are expected to enforce the separation of the worlds. However, the secure world's RAM partition is not encrypted, leaving it vulnerable to physical attacks.

In contract to Intel SGX, TrustZone does not offer an isolated environment for each individual enclave. Instead, the secure world is shared by all enclaves. The secure world also contains a dedicated OS, which further increases the TCB size. Out of the box, TrustZone does not offer remote attestation, remote provisioning, or sealing.

AMD SEV-SNP. *AMD Secure Encrypted Virtualization* (AMD SEV) [60] enables the isolated execution of VMs under the assumption of a "benign but vulnerable" hypervisor[12]. AMD SEV tags all cache-resident code and data of a protected VM and restricts access to only the VM corresponding to the tag. It further uses a dedicated on-chip security subsystem called *AMD secure processor* (AMD-SP) for key management. Before an isolated VM starts, the AMD-SP randomly generates a memory encryption key for the VM, shares the key with an on-chip

12 For AMD, this means that the hypervisor could have exploitable vulnerabilities, but does not actively try to compromise the protected VMs [107].

memory controller, and isolates the key from all software processed by the CPU. Whenever data leaves or enters the CPU's caches, the memory controller uses the tag to identify the corresponding memory encryption key and transparently encrypts and decrypts the data. The tag-based access control and memory encryption with individual keys enforce isolation between the protected VMs.

Applications executed inside a protected VM do not require any modification, and the AMD-SP provides an API for remote attestation and remote provisioning. However, an application's code and data are not protected against other software running inside the VM. In particular, the whole guest OS is part of the TCB.

As presented initially in 2016, AMD SEV did not provide memory integrity and freshness guarantees. In 2017, AMD introduced an extension called AMD SEV-ES [108]. This extension encrypts the VM's registers with the VM's memory encryption key and computes an integrity-check value whenever the VM is stopped (due to an interrupt or another event). Once the VM is resumed, SEV-ES decrypts and integrity checks the registers before restoring them. In 2020, AMD presented a further AMD SEV extension called AMD SEV-SNP [107]. This extension adds memory integrity and freshness protection to data stored outside of the CPU, e.g., in RAM or on disk. With AMD SEV-SNP, the hypervisor is considered fully untrusted. Note that AMD SEV-SNP was not available while the research for this dissertation was performed.

IBM SE. Recently, IBM announced the availability of IBM SE [109], [110], a TEE for IBM z15 and LinuxONE III generation systems. Comparable to AMD SEV, IBM SE protects VMs from being inspected or modified by other software. As with AMD SEV, modifications of the applications running in the VM are not necessary. A difference to AMD SEV is that IBM SE loads an encrypted VM image into the isolated environment, enabling offline provisioning of sensitive data, e.g., passwords, hashes, or private keys.

To achieve offline provisioning, every server has a private host key, which is only accessible to IBM hardware and a trusted firmware called the *ultravisor*. The cloud provider receives a host key document, which is signed by IBM and contains the host's public key. The cloud provider can forward the host key document to a customer, who verifies it using a PKI. Afterwards, the customer creates a random image key, uses the image key to encrypt a VM image, and creates an IBM SE header. This header is integrity protected, contains a cryptographic hash of the image, and contains the image key encrypted with the host's public key. After the customer deploys the header and the encrypted image at the cloud provider, ultravisor verifies the integrity of the header, verifies the integrity of the image, decrypts the image using the host's private key, and starts the image.

IBM SE provides confidentiality and integrity protection for all pages in the "secure memory". Every page in this memory is tagged with a VM id and they can only be accessed by the corresponding VM and the ultravisor. The state of the VM, e.g., CPU registers, cryptographic keys, and program status words, are also protected by the ultravisor. However, publicly available documents do not specify if this secure memory is encrypted and thus protected against physical attacks. The support of remote attestation, remote provisioning, and a source of randomness is also not specified.

4

Related Approaches for Secure, Outsourced Data Processing

As we describe in Section 1.1, this dissertation explores how outsourced data structures are processed in the secure, outsourced, TEE-based data processing scenario, i.e., a data owner wants to process its sensitive data at an untrusted cloud provider, which uses a TEE for the processing of data structures. In this chapter, we differentiate our work from related approaches that allow secure, outsourced data processing (with and without a TEE). Ideally, the approaches should combine strong security, high efficiency, and arbitrary processing capabilities. We show that many related works aim to achieve this goal, but fail in at least one of these aspects or they cover a different scenario.

We begin with a brief description of the following approaches: *encrypted outsourced storage*, *secure multi-party computation* (MPC), and *fragmentation*. Each description includes an introduction to the respective scenario. Afterwards, we present the following approaches: *property-preserving encryption* (PPE), *searchable encryption* (SE), *homomorphic encryption*, and *functional encryption* (FE). These approaches use the *secure, outsourced data processing* scenario, which we describe in the next paragraph. Finally, we discuss approaches using the secure, outsourced, TEE-based data processing scenario, but do not process data structures inside an enclave. In all cases, we end the approach description with a brief assessment stating why the approach does not fulfill the requirements of this dissertation.

The secure, outsourced data processing scenario (see Figure 4.1) works as follows: A trusted data owner encrypts its data locally and outsources the ciphertexts to an untrusted cloud provider. The data owner then sends requests to the cloud provider, which processes the encrypted data and sends a response. Request, processing, and response depend on the approach and we explain details in the corresponding sections. The difference to our scenario is that the cloud provider does not use a TEE for data processing. Outsourced processing without a TEE would be preferable, as a TEE introduces additional trust assumptions. However, security, efficiency, and/or processing capabilities are a problem for all existing approaches without a TEE.

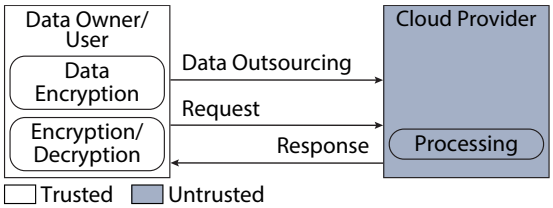

Figure 4.1: Secure, outsourced data processing scenario.

Note that the three main chapters of this dissertation each have a dedicated related work section. These sections discuss approaches applicable to the secure, outsourced processing of the specific data structure considered in the corresponding chapter.

4.1 Encrypted Outsourced Storage

We denote the most straightforward approach for processing of outsourced data by *encrypted outsourced storage* and it works as follows: In a setup phase, the data owner encrypts its data using an SKE (or an AE) scheme and outsources the encrypted data to a cloud provider. In the runtime phase, the data owner downloads its data, decrypts it, and performs queries on plaintext data. The encrypted outsourced storage approach perfectly protects the confidentiality (and integrity) of the outsourced data, and it allows arbitrary data processing. Figure 4.2 illustrates the outsourcing scenario used in this case.

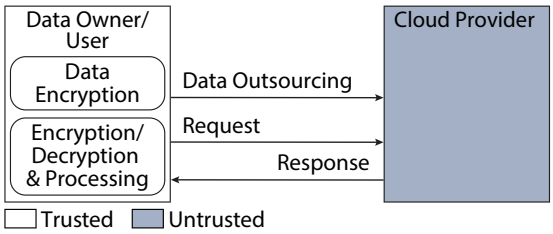

Figure 4.2: Outsourcing scenario used by encrypted outsourced storage.

Assessment. Technically speaking, the encrypted outsourced storage approach does not provide outsourced *data processing*. Instead, the data owner has to perform the computing-intensive processing. The data transfer is very inefficient and a main benefit of cloud computing—cost efficient processing—is not leveraged by this approach. We still mention encrypted outsourced storage as it is the most straightforward approach, provides the highest level of security, and supports arbitrary processing capabilities (at the data owner).

4.2 Secure Multi-party Computation (MPC)

Secure multi-party computation (MPC) enables a group of mutually distrustful data owners to jointly compute a publicly known function over their private inputs. An MPC protocol is considered secure if the data owners only learn the function's output (and the information that can be inferred by the output). Yao [111], [112] introduces MPC for two data owners and Goldreich *et al.* [113] extend it to multiple data owners. Since then, researchers made many improvements, e.g., they introduced a formal security definition [114], explored the number of necessary communication rounds [115], and improved the performance [116]. There are three common MPC scenarios:

1. In the *distributed* MPC scenario, multiple data owners perform an interactive protocol to compute the public function without any third party (see Figure 4.3a).
2. In the *server-aided* MPC scenario, an untrusted cloud server provides its computational resources for the function evaluation, but does not contribute own inputs to the function (see Figure 4.3b). As a result, the users are relieved (partially) from computational-intensive processing. Depending on the approach, the server is [117] or is not allowed to learn the output [118].
3. In the *multiple-servers* MPC scenario, multiple cloud servers evaluate the function on the data owners' inputs without contributing own inputs to the function (see Figure 4.3c). The benefit over the server-aided scenario is that the processing is secure, even if multiple data owners and a specific number of servers (depending on the approach) collude [119].

For more details about MPC including state-of-the-art, we refer to a paper from Evans *et al.* [120].

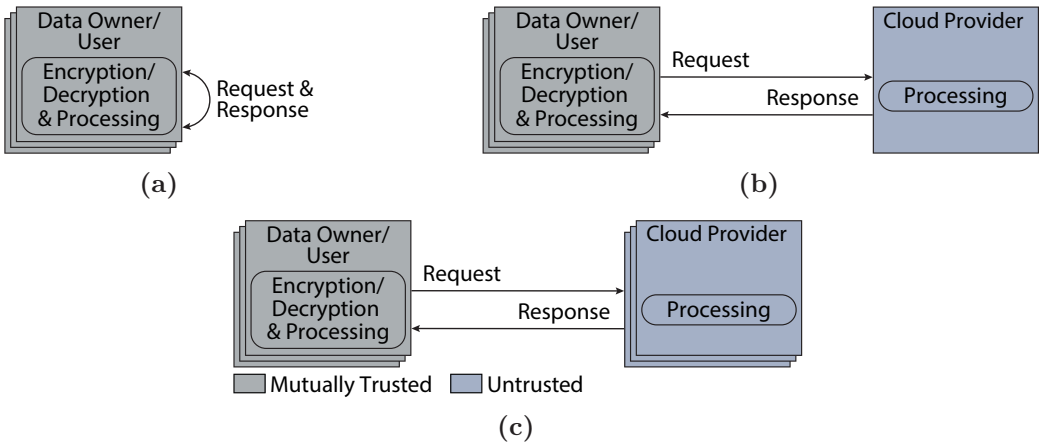

Figure 4.3: Outsourcing scenarios used by MPC: (a) distributed, (b) server aided, and (c) multiple servers.

Assessment. MPC is beyond the scope of this dissertation as all three MPC scenarios fundamentally differ from our scenario variants, which only consider a single data owner. Furthermore, most MPC approaches are inefficient in a distributed cloud environment as they require multiple communication rounds and/or the size of transferred messages depends on the complexity of the computed function.

4.3 Fragmentation

Aggarwal *et al.* [121] introduce a combination of data fragmentation and encryption. For a database table containing multiple attributes, the linkage of these attributes is protected by distributing them to two non-colluding cloud providers and encrypting the attributes if necessary. Other researchers extend the fragmentation to more than two cloud providers [122]–[124]. For each request, the data owner needs to carefully rewrite its request to query data from the appropriate cloud provider(s). For each response, the data owner needs to postprocess the data to receive the final result. Figure 4.4 illustrates the scenario used in this case.

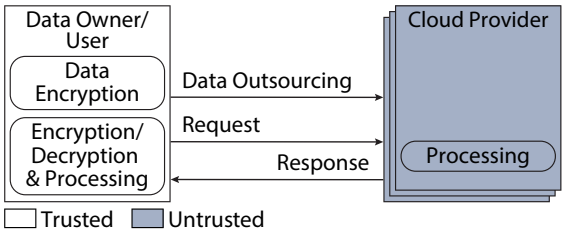

Figure 4.4: Outsourcing scenario used by fragmentation approaches.

Assessment. Fragmentation approaches have two main problems: (1) The protection only holds if two or more cloud providers do not collude, which cannot be guaranteed in practice. (2) The query rewriting and postprocessing introduce an additional level of processing complexity at the data owner.

4.4 Property-preserving Encryption (PPE)

The encrypted outsourced storage approach is inefficient, because the encryption schemes used protect all information about the underlying plaintext data. The cloud provider is untrusted; thus, cannot receive the encryption key and cannot support the processing. The idea of *property-preserving encryption* (PPE) is to preserve some properties of the underlying plaintext data in the corresponding ciphertexts. A cloud provider can use the preserved properties to process encrypted data without decryption. In the following subsections, we present two PPE schemes: *deterministic encryption* (DET), which preserves the equality relation, and *order-preserving encryption* (OPE), which preserves the order relation.

4.4.1 Deterministic Encryption (DET)

A DET scheme preserves the equality relation of plaintext values, i.e.,

$$\mathsf{DET\_Enc}(SK, V) = \mathsf{DET\_Enc}(SK, U) \iff V = U$$

with DET_Enc denoting the encryption algorithm of a DET scheme. In the secure, outsourced data processing scenario (see Figure 4.1), the data owner can request equality operations, e.g., existence, count, and equality selection queries. The cloud provider performs these operations on ciphertexts and sends back the result of the query. In the symmetric key setting, block ciphers (without a random initialization vector) can be used to implement a DET scheme, and in the public-key setting, multiple approaches exist [10], [125].

Security assessment. The equality of encrypted values, however, is also the main downside of DET schemes as it inherently leaks the frequency of underlying plaintext values—even without user interaction. Naveed *et al.* [13] show that this inherent leakage can be used for a highly effective frequency analysis attack, e.g., for 200 encrypted hospital databases with small data domains (at most 365 distinct values), the attacker recovers the disease severity for 100% of the patients for 51% of the databases.

A possible security definition for DET schemes is IND-DCPA security [10], [126], which is defined with the following experiment and probability: A challenger randomly picks a bit b, the attacker queries pairs $(V_1, U_1), \ldots, (V_q, U_q)$ with distinct $V_1, \ldots, V_q$ and distinct $U_1, \ldots, U_q$, and the challenger returns $\mathsf{DET\_Enc}(SK, V_i)$ if $b = 0$ and $\mathsf{DET\_Enc}(SK, V_j)$ otherwise. Compared to a random guess, the attacker's probability to guess b should be negligible. Stronger security definitions might be possible, but due to the inherent leakage, a DET scheme cannot be IND-CPA secure.

4.4.2 Order-preserving Encryption (OPE)

An OPE scheme preserves the order relation of plaintexts, i.e.,

$$\mathsf{OPE\_Enc}(SK, V) \leq \mathsf{OPE\_Enc}(SK, U) \iff V \leq U$$

with OPE_Enc denoting the encryption algorithm of an OPE scheme. In the secure, outsourced data processing scenario (see Figure 4.1), the data owner can request order operations, e.g., range, sort, and rank queries. The cloud provider performs these operations in logarithmic runtime (in the number of ciphertexts) on the ciphertexts and sends back the result of the query.

Agrawal *et al.* [11] introduce OPE and propose an OPE construction. However, the data owner needs to know the plaintext distribution before encryption and a formal security analysis is missing. Boldyreva *et al.* [12] state that OPE schemes cannot achieve IND-CPA security

and introduce two new security definitions: IND-OCPA and the weaker POPF-CCA. An IND-OCPA–secure OPE scheme leaks only the order of the underlying plaintexts. Boldyreva *et al.* prove that such a scheme requires an exponentially large ciphertext space (in the size of the plaintext space) if ciphertexts are immutable and the scheme is stateless. Furthermore, they present a POPF-CCA–secure OPE scheme, but it does not hide the distance between plaintexts and leaks at least half of the plaintext bits [127]. Popa *et al.* [128] present the first IND-OCPA–secure OPE scheme achieving linear-length ciphertexts by using an interactive multi-round protocol, ciphertext updates, and server-side state (linear in the number of distinct plaintexts). Kerschbaum *et al.* [129] improve the idea of Popa *et al.* reducing the probability of computationally expensive ciphertext updates and the number of rounds. However, the scheme requires client-side state (linear in the number of distinct plaintexts). Kerschbaum [130] presents a randomized OPE scheme, i.e., duplicate plaintexts are encrypted to different ciphertext. The scheme is IND-FAOCPA secure, which is strictly stronger than IND-OCPA security as it hides the plaintext frequency. However, it requires even more client-side state than the scheme from Kerschbaum *et al.* [129].

MOPE. Boldyreva *et al.* [127] propose an OPE extension called *modular OPE* (MOPE). The idea is to add a fixed, secret offset to all plaintext before OPE encryption and to carefully rewrite queries to enable (modular) range queries. An MOPE scheme hides the location of encrypted values, but its security falls back to POPF-CCA security if the attacker learns the secret offset. Mavroforakis *et al.* [131] design protocols to mitigate the probability of secret offset leakage.

ORE. Boldyreva *et al.* [127] also propose *order-revealing encryption* (ORE), a generalization of OPE. ORE schemes assume a publicly computable function that returns the order of two ciphertexts. Boneh *et al.* [132] present a stateless, non-interactive, IND-OCPA–secure[1], but inefficient ORE scheme. Chenette *et al.* [133] present an efficient ORE scheme, but it leaks the first bit at which two ciphertexts differ. Lewi *et al.* [134] present two efficient ORE schemes: The first is IND-OCPA secure, but only works for a small plaintext space. The second supports large domains, but leaks the block[2] at which two ciphertexts differ.

Security assessment. As can be seen from this non-exhaustive list of OPE and related approaches, there is a plethora of research, but security definitions and assumptions are still debatable. The practical consequences of OPE's leakage are difficult to capture as it depends on the attacker's side knowledge and the data distribution.

In the following, we say that a database is *dense* if the encryption of each plaintext value is contained at least once. With a dense database, all deterministic OPE and ORE schemes[3] are susceptible to a simple attack: map the sorted ciphertexts one-to-one to sorted plaintexts. With a non-dense database, Naveed *et al.* [13] empirically explore the security of IND-OCPA–secure, deterministic OPE and ORE schemes. Their attack uses a dump of encrypted data, publicly available auxiliary information, the order leakage, and the frequency leakage. Tested on 200 encrypted hospital databases with small data domains (at most 365 distinct values), the attack recovers more than 80% of the patient records for 95% of the databases. Durak *et al.* [135] use correlation between multiple columns to perform a successful attack on location data that is sparse in the plaintext domain and only contains unique values. In a dataset containing the encrypted latitude and longitude 21,000 intersections in California, the attack reconstructs the location with a precision between 2 and 140 km. Grubbs *et al.* [14] present an attack that uses auxiliary information to reliably recover high-frequency elements, even if the frequency is not

1 Technically, IND-OCPA security is only defined for OPE, but we mean a straightforward adaptation.
2 A block consists of one or more bits.
3 This only excludes Kerschbaum's randomized OPE scheme [130].

leaked. Thus, this attack is also successful against randomized OPE schemes, e.g., it recovers 30% of first names from a customer record database encrypted with Kerschbaum's scheme [130]. Lacharité *et al.* [15] propose to consider a persistent, passive adversary, which uses rank leakage (induced by all schemes discussed before), uniformly distributed range queries, and access pattern leakage, i.e., the set of values matching a query. In a dense database, the attack is able to reconstruct all plaintexts after $N \log(N) + O(N)$ queries where N is the number of distinct values. Kellaris *et al.* [136] introduce so-called volume attacks. In these attacks, the adversary only learns how many ciphertexts are returned by the server and uses this information to recover which plaintexts are encrypted how often. Under the assumption of uniform range queries, a maximum of 150 ciphertexts, and $O(|\mathcal{D}|^4 \log(|\mathcal{D}|))$ observed queries with $|\mathcal{D}|$ being the domain size, their attack can reconstruct all plaintexts for all tested datasets. Grubbs *et al.* [137] and Gui *et al.* [138] present further volume attacks, which are successful under weaker assumptions and with fewer queries.

4.4.3 Assessment

DET and OPE support an important set of operations on encrypted data and the efficiency is high for these operations. However, as we explained in detail in the last two sections, the security of both PPE schemes is debatable.

4.5 Searchable Encryption (SE)

Searchable encryption (SE) enables a data owner to outsource documents[4] to an untrusted cloud provider, while preserving the ability to retrieve documents containing one or multiple keywords. SE works as follows in the secure, outsourced data processing scenario (see Figure 4.1): After ciphertext outsourcing, the data owner uses a secret key to generate a search token, which he sends to the cloud provider. The cloud provider uses the search token to unveil for each document whether it contains the searched keyword(s).

The goal of SE is to leak nothing beyond the outcome of the search and the pattern of a sequence of searches. In particular, without a search token the ciphertexts should be IND-CPA secure. Thus, SE achieves a stronger notion of security than PPE, because PPE inherently leaks the preserved information about the underlying plaintext data, even without a single request. In the following, we explore SE schemes supporting two different query types: First, SE schemes supporting keyword searches, i.e., the cloud provider can use the search token to determine if a document contains a keyword. Second, SE schemes supporting range searches, i.e., the cloud provider can use the search token to determine if a document contains a keyword in a certain range. For both query types, we discuss approaches that do and do not use an index. For an in-depth introduction and other search types, we refer to a survey from Bösch *et al.* [139].

Keyword searches without an index. Song *et al.* [140] introduce symmetric SE and propose a scheme for exact keyword searches. Depending on an adjustable parameter, the scheme returns a low or high number of false positives. Boneh *et al.* [141] propose a public-key SE scheme that allows any party knowing the public key to encrypt keywords, but only secret key holders can generate search tokens. Due to the computation cost of public key encryption, the scheme is only applicable on a small number of keywords. A downside of both schemes mentioned so far is that their search time is linear in the size of all ciphertexts.

4 The term documents is used generically and can mean, e.g., text documents, emails, or audit logs.

Keyword searches using an index. Goh [142] proposes the use of an inverted index to build an SE scheme with a search time linear in the number of indexed documents. Additionally, he formulates the first security definition for secure search indices (IND-CKA), which demands that besides the knowledge gathered from previous queries, the content of documents should not be revealed by the index and from other channels. Goh's SE scheme uses bloom filters inducing false positives. Chang *et al.* [143] also propose an index-based SE scheme. This scheme uses pseudorandom bits to mask entries in a prebuilt directory, does not have false positives, and achieves a stronger security definition (IND2-CKA), which additionally hides the document sizes. However, the scheme requires two rounds and thus is less efficient than Goh's scheme. Curtmola *et al.* [144] introduce a new attacker model differentiating adaptive and non-adaptive attackers, i.e., the adversary can or cannot choose her queries based on the previously obtained search tokens and results. They state that all SE schemes presented before are only secure for non-adaptive adversaries and present an SE scheme secure in an adaptive adversary setting. In this scheme, the search token and the server-side storage are linear in the size of the largest document.

Range searches without an index. Boneh *et al.* [145] propose an SE scheme supporting range searches in the public-key setting. Their scheme is inefficient as the ciphertext and public key size are linear in the plaintext domain. Shen *et al.* [146] propose an SE scheme in the private-key setting. The scheme supports inner-product queries, which can also be used for, e.g., conjunctive, disjunctive, and exact threshold queries. However, the ciphertexts produced by the scheme are also linear in the plaintext domain. Shi *et al.* [147] present an SE scheme in the public-key setting. This scheme has a ciphertext and public key size logarithmic in the plaintext domain, but it leaks the plaintext of matching ciphertexts. The main issue of the SE schemes by Boneh *et al.*, Shen *et al.*, and Shi *et al.* is that they have linear search time (in the number of documents).

Range searches using an index. Lu [148] combines the SE scheme from Shen *et al.* [146] with an index tree to achieve polylogarithmic search time. However, his index tree inherently reveals the order of the ciphertexts, making it vulnerable for the attacks presented for OPE. Demertzis *et al.* [149] present an SE scheme (Logarithmic-URC) that does not leak the ciphertext order and improves the constant factor of a range search.

Assessment. As we have seen in this section, SE schemes provide a wide range of processing capabilities and have strong security definitions. The main issue with SE is that a search is orders of magnitude slower than TEE-based approaches. For instance, if the scheme from Demertzis *et al.* [149] is used to encrypt 100,000 documents, a range search for 100 keywords takes more than a second (see Section 6.6.2). A corresponding TEE-based search implemented by us takes 0.125 ms.

4.6 Homomorphic Encryption

Fully homomorphic encryption (FHE) supports a set of operations, e.g., addition and multiplication, on ciphertexts without decryption, i.e.,

$$\mathsf{FHE\_Dec}(SK, \mathsf{FHE\_Enc}(SK, V) \otimes \mathsf{FHE\_Enc}(SK, U)) = V \odot U$$

where $\odot$ is an operation from this set in the plaintext domain and $\otimes$ not necessarily the same operation on the ciphertexts. Using a composition of these operations, the data owner can request the execution of an arbitrary function F (also called circuit) on ciphertexts in the secure, outsourced data processing scenario (see Figure 4.1). The cloud provider executes F

and returns the encrypted result. *Fully homomorphic encryption* (FHE) provides semantic security for stored encrypted values, during processing, and for the results.

Rivest *et al.* [16] proposed the idea already in 1978, but only in 2009, Gentry [17] presented the first instantiation of FHE. Since then, Gentry *et al.* [18], Halevi *et al.* [150], Chillotti *et al.* [151], and others present improved FHE schemes.

PHE. Goldwasser *et al.* [152] introduce *partial homomorphic encryption* (PHE) schemes. These schemes allow only one specific arithmetic operations on encrypted data, e.g., addition [153], multiplication [154], and bitwise XOR [152], with an improved efficiency compared to FHE.

PHE-TEE hybrid. Quite recently, Fischer *et al.* [46][5] proposed to combine PHE schemes with a TEE. If data is encrypted with a PHE scheme supporting the required operation, the operation is executed on encrypted data. Otherwise, a TEE is used to decrypt the data and encrypt it to the required PHE scheme, before the operation is executed. Consequently, multiple operations can be supported with different PHE schemes.

Assessment. FHE offers an optimal solution for the secure, outsourced data processing scenario regarding security and processing capabilities. However, FHE is too inefficient for adoption in the systems we consider in this dissertation. PHE schemes are more efficient, but they only support one operation on encrypted data; thus, they are insufficient for the complex outsourced data processing we target. For the PHE-TEE hybrid approach, it is an open question to determine the security, efficiency, and processing capabilities achievable if it is used for large applications in the secure, outsourced data processing scenario.

4.7 Functional Encryption (FE)

As with FHE, *functional encryption* (FE) allows the evaluation of an arbitrary function F on ciphertexts. FE is even more powerful than FHE, because it can reveal the result of F and not only a ciphertext of the result. In the secure, outsourced data processing scenario (see Figure 4.1), FE is used as following: After the ciphertext outsourcing, the data owner uses a secret key to generate a token for a specific function F. The data owner (or a user, who received the token) sends the token to the cloud provider. For each ciphertext C with underlying plaintext value V, the cloud provider uses the token to evaluate $\mathsf{F}(V)$ and returns the results back to the data owner (or user). The only information leaked to the cloud provider is the result of $\mathsf{F}(V)$.

Sahai *et al.* coined the term functional encryption in a presentation [155]. O'Neill [156] and Boneh *et al.* [157] formally introduce the general concept of FE (in a public-key setting). Goldwasser *et al.* [158] extend the general concept to multi-input functions, i.e., the data owner sends a token, which the cloud provider uses to reveal $\mathsf{F}(V_1, \ldots, V_n)$ for n ciphertexts $C_1, \ldots, C_n$ with underlying plaintext values $V_1, \ldots, V_n$.

Special cases of FE. In the following, we briefly describe three special cases of FE and we note that most of them are older than the term functional encryption:

1. In searchable encryption, the data owner outsources its encrypted documents, generates search tokens, and sends the tokens to the cloud provider [140], [141]. Using the search token, the cloud provider evaluates F to unveil for each document whether it matches the search and sends back the matching documents.
2. In attribute-based encryption, the data owner assigns attributes to data, outsources the encrypted data together with the attributes, and generates access tokens for users [159]–

5 The author of this dissertation is co-author of this paper.

[161]. Using their access tokens, users can recover plaintext data if the access control check done by F grants the access.
3. Some order-revealing encryption schemes can be considered a special case of multi-input functional encryption, at which F returns the order relation of ciphertexts [132], [134].

Assessment. FE combines strong security and arbitrary processing capabilities. Yet, the main problem of FE is that it is too inefficient for practical systems. This is especially true for generic FE constructions [162], [163], but is also true for the cryptographic systems that can be considered a special case of FE.

4.8 TEE-based Approaches

All related approaches presented so far only rely on software for secure, outsourced data processing. Now, we explore approaches in the secure, outsourced, TEE-based data processing scenario, i.e., the cloud provider can use a TEE for (parts of) the outsourced processing.

In this scenario, the granularity of the processed object, which is protected by the TEE, is flexible. The granularity ranges from the protected processing of entire applications to the protected processing of individual, stateless operations. In this section, we first describe approaches from both ends of this range and discuss their advantages and disadvantages. Finally, we provide an assessment of both approaches and briefly introduce the trade-off that we use in this dissertation.

4.8.1 Protect Entire Applications

The following approaches protect the execution of entire (legacy) applications using Intel SGX. The goal is to reduce the necessary modification to unprotected application versions as far as possible. To achieve this goal, the approaches wrap the applications with *auxiliary software* of varying extent to simulate a regular OS.

Baumann *et al.* [164] propose Haven, an approach which uses an in-enclave, Drawbridge-based [165] library OS to execute unmodified Windows application as an enclave. The library OS supports secure variants of OS functionality in the enclave, e.g., confidentiality- and integrity-protected file I/O, user-level threading, and secure virtual memory management. Haven simulates the full OS API to the enclave by handling many tasks in the enclave and calling a narrow set of primitives at the host OS. Baumann *et al.* only evaluate Haven with a simulated TEE, because the paper predates the availability of Intel SGX–enabled CPUs. Haven's auxiliary software has millions of LOC and 209 MB.

Tsai *et al.* [166] present Graphene-SGX, an approach which uses an in-enclave, Graphene-based [167] library OS to execute unmodified Linux applications as an enclave. As done by Haven, the library OS provides many OS functions inside the enclave. It additionally supports enclave-level forking and secure inter-process communication. Graphene-SGX restricts the application's resource access using a manifest, which is specified and signed by the developer. The manifest can also be used by the developer to provide hashes of trusted files and to define untrusted output files and directories. Tested on three Linux web servers, the peak throughput of Graphene-SGX is 26–66% lower compared to native implementation. Graphene-SGX's auxiliary software has about 1,348,000 LOC.

Arnautov *et al.* [67] propose SCONE, a container mechanism for Intel SGX enclaves using Docker. Instead of an in-enclave library OS, SCONE uses an in-enclave C library. To cater for the missing OS functionality, SCONE delegates the calls to the host OS instead of emulating them inside the enclave. To achieve efficient processing, it uses an in-enclave, asynchronous system call interface and a SCONE kernel module outside of the enclave. Like Haven, SCONE

supports confidentiality- and integrity-protected file I/O and user-level threading. Furthermore, it offers encrypted console streams and a TLS endpoint inside the enclave for secure network communication. After static recompilation of four small applications fitting into the EPC, the peak throughput of SCONE is between 40% lower and 20% higher compared to native implementations[6]. SCONE's auxiliary software has about 187,000 LOC.

Shinde *et al.* [168] propose PANOPLY, a mechanism for micro-containers in enclaves. These micro-containers expose OS functionality such as file system access, network access, multithreading, forking, and thread synchronization to unmodified applications. As SCONE, PANOPLY delegates OS functions to the host OS instead of emulating them inside the enclave. Opposed to SCONE, PANOPLY offers applications a synchronous POSIX-level interface instead of an asynchronous system call interface. As a result, PANOPLY does not require any libc library in the enclave. Furthermore, PANOPLY enables the developer to partition an application into multiple enclaves using annotations. Tested on four applications, PANOPLY introduces an average performance overhead of 24%, and the authors report a 5–10% higher overhead compared to Graphene-SGX. PANOPLY's auxiliary software has about 20,000 LOC.

Discussion. It would be ideal for developers to put entire applications into an enclave, because it saves the refactoring overhead. However, we see three problems with this approach:

1. Recall that the TCB of an Intel SGX–enabled application has only one flexible component—the enclave size (see Section 3.1.5). The auxiliary software of the presented approaches alone have thousands to millions LOC. Together with the potentially large (legacy) applications, this opens up a huge attack surface. We agree with a statement given by Intel in the Intel SGX developer reference [24]: "The application writer should make the trusted part as small as possible. It is suggested that enclave functionality should be limited to operate on the secret data. A large enclave statistically has more bugs and (user created) security holes than a small enclave. [...] Embracing the above design considerations will improve protection as the attack surface is minimized." Besides the problems mentioned by Intel, the following issues become more problematic with a growing enclave size: the data owner might want to (formally) verify the enclave code and an attacker can try to hide malware inside an enclave.
2. Approaches using a library OS such as Haven and Graphene pull most OS functionality into the enclave. This reduces the size and complexity of the enclave's interface, but (vastly) increases the TCB. Other approaches such as SCONE and PANOPLY use more delegated system calls to reduce auxiliary software in the enclave. This decreases the TCB, but increases the size and complexity of the interface. A critical aspect of both ideas is that the original applications were designed under the assumption of a trusted OS. Therefore, the developers did not worry about the leakage of calls to the OS. The auxiliary software protects some calls, but it is very difficult to determine the leakage of all calls across the enclave boundary, especially for large applications.
3. In the current Intel SGX version, the EPC has only about 96 MB. With a high probability, the application's code and data, together with the auxiliary software, exceed this restricted memory space. This causes paging and with it a devastating performance penalty.

4.8.2 Protect Individual, Stateless Operations

The following approaches use TEEs[7] to protect the processing at the minimal granularity—the processing of individual, stateless operations. The two approaches that we present use secure,

6 Arnautov *et al.* do not explain the increased peak throughput.

7 Strictly speaking, the approach by Arasu *et al.* [169] uses an FPGA to evaluate isolated execution. This was done, because TEEs were not yet widely available.

operator-based processing in an encrypted *database management system* (DBMS) setting. They offer different security levels on a column basis and allow processing across different levels. As we are only interested in the implementation of the idea, we abstract from the setting and assume that generic values are encrypted on the highest security level.

Arasu *et al.* [169] present Cipherbase, which uses a dedicated, custom-designed FPGA for the secure processing of individual, stateless operations. The authors propose to encrypt every data value individually and to change every operation invocation to the following process: transfer the corresponding encrypted data to the FPGA, decrypt the data in the FPGA, perform the operations in the FPGA, and return the (encrypted) result. Cipherbase supports various primitives, e.g., equality comparisons, arithmetic operations, and aggregations. For all boolean operations, the result is plaintext. The decryption keys are derived from a master key, which has to be provisioned to the FPGA securely. As FPGAs do not support remote attestation and remote provisioning, this has to be done in a secure environment before the FPGA is shipped to the cloud provider, which is unsuitable for (large scale) cloud deployment (see Section 3.3). In a follow-up paper [170], the authors present performance optimizations. Due to the high round-trip latency between CPU and FPGA, and the low processing power of the FPGA, Cipherbase still reduces the throughput of a protected DBMS to 40% compared to an unprotected version. The authors briefly mention Intel SGX, but leave an adaptation of Cipherbase open for future work.

Vinayagamurthy *et al.* [29] present StealthDB, an encrypted database using Intel SGX. Comparable to Cipherbase, the authors propose to encrypt every data value individually and to change every operation invocation to a call to an Intel SGX enclave. StealthDB supports arithmetic operations (e.g., $+, -, *$), relational operations (e.g., $<, >, <>$), boolean operations (e.g., AND, OR, NOT), hash functions, and math functions (e.g., sin, cos, tan). The operations are data-oblivious to protect against side-channel leakage.

Discussion. On the one hand, the approaches presented adhere to Intel's recommendation to keep the enclave as small as possible and to limit the operations on secret data. As the set of low-level primitives is limited, the enclave can have a constant size and can be used generically for various applications. As a result, application protected with these approaches have a small attack surface, a low likelihood of bugs, and a low interface complexity. Compared to a large enclave, it is easier for the developer to protect against attacks, easier for the data owner to verify the enclave code, and harder for an attacker to hide malware inside an enclave (see Section 3.2). It might even be possible to build a compiler, which automatically transforms applications. Such a compiler could reduce the development overhead for applications designed for a TEE environment and it could reduce the refactoring overhead for legacy applications.

On the other hand, the approaches presented inherently leak the (encrypted) result of each operation to a persistent attacker. It is difficult to capture the exact leakage, because it depends on the processed operation and the operation's position in the code. For instance, a data search using relational operations leaks the relation of all processed encrypted data; the plaintext result of a boolean operation might leak plaintext of the involved encrypted values; and the input and result sizes of an arithmetic operation might leak a value range of the encrypted values. Furthermore, the approaches leak the exact path taken through the application code, which might inherently leak sensitive information. Another downside is the high number of calls to the TEE, which introduces a non-negligible overhead.

4.8.3 Assessment

As a multitude of Intel SGX-based papers [171] and this dissertation show, TEE-based solutions enable arbitrary data processing combined with a high efficiency. The processing is also secure

if the TEE concept is implemented flawlessly by the vendor. In Section 3.2, we give an extensive overview of attacks on Intel SGX and mitigations for these attacks. Our main observation is that the enclave size should be kept small. Besides that, it is up to a user to evaluate whether the security guarantees are strong enough.

In the last two sections, we explained why it is problematic to put entire applications or individual, stateless operations into an enclave. In summary, the downsides of the entire application approach are that the TCB is large, it is difficult to capture the exact leakage, it impedes attack mitigations (see Section 3.2), and the enclave might not even fit into available TEEs. The individual, stateless operations approach does not suffer from these problems, but it leaks the result of each operation to a persistent attacker.

In this dissertation, we use a trade-off between both approaches—the TEE-based processing of data structures. We introduce this approach and explain its advantages and disadvantages in Section 5.1.

5 Methodology

In this chapter, we first motivate and describe the design principles that we use for the approaches presented in this dissertation. Then, we present our security and performance assessment methodology, which we use in the following chapters to answer our research question (see Chapter 1.1): "For outsourced systems using a memory-limited, widely available trusted execution environment (TEE) to process data structures at an untrusted cloud provider, what are lower bounds of security and corresponding upper bounds on performance?"

5.1 Design Principles

As we show in Section 4.8, the secure, outsourced, TEE-based data processing scenario offers a favorable combination of strong security, high efficiency, and arbitrary processing capabilities, compared to other known approaches allowing secure, outsourced data processing. However, we also show that it has severe downsides to put entire applications or individual, stateless operations into an enclave.

We draw three lessons from the observations in Section 4.8: (1) we want to use a TEE at the cloud provider for secure, outsourced data processing, (2) we want to keep the enclave size as small as possible, and (3) we do not want to leak the result of every individual decision. We adhere to these lessons using a trade-off between both approaches presented in Section 4.8—we use a TEE for the processing of data structures. In more detail, we perform self-contained data structure operations in our enclaves and only return the result, e.g., use one ECall (see Section 3.1.2) to perform a range search in a B^+-tree, an equality search in a dictionary, or an access control check in a file system.

The following example highlights the difference between the individual, stateless operations approach and the data structure approach: We assume that an array contains encrypted values and an enclave should be used for a binary search. In the individual, stateless operations approach, the algorithm sends two encrypted values to the enclave, receives the plaintext result of the comparison, and repeats this process until the search is done. In the data structure approach, the algorithm sends the whole array to the enclave at once and only receives the result of the search.

Our data structure approach requires more enclave code than the individual, stateless operations approach, because the processing is more complex. However, it requires considerably less enclave code than the entire application approach with its auxiliary software, because we keep all system code that does not process sensitive data, outside of the enclave. Consequently, our enclave size is small, and we share many benefits with the individual, stateless operations approach, e.g., a small attack surface, a low likelihood of bugs, a low interface complexity, and a facilitation for code verification by the data owner. As we show in the section about attacks on Intel SGX (Section 3.2), a small enclave size is key for attack mitigation. A downside of our approach is that it cannot be used as a generic solution to protect (legacy) application. Instead, we design our enclaves specifically for a given problem.

We expect that large datasets are processed in the secure, outsourced, TEE-based data processing scenario. Yet, Intel SGX has only about 96 MB of EPC memory and during the timeframe in which the research for this dissertation was done, it was the only TEE fulfilling the required capabilities (see Chapter 3). If we load data into an Intel SGX enclave exceeding 96 MB, Intel SGX starts paging (see Section 3.1.1). Alternatively, we can store encrypted data in untrusted memory and let the enclave load the data on demand in smaller chunks. In the evaluation section of Chapter 6, we compare these alternatives and show that paging leads to severe performance degradation and on demand loading should be preferred.

To achieve a high performance when loading data into the enclave on demand, it is important to have a high throughput. To achieve this goal, Intel SGX enclaves can access the memory of its host application to prevent expensive copy instructions (see Section 3.1.2) and Intel SGX provides switchless calls to prevent context switches (see Section 3.1.7).

Overall, we adhere to the following design principles:

- Use a TEE for the secure, efficient processing of outsourced data.
- Keep the enclave size small.
- Use enclaves to process data structures.
- Store data in the untrusted environment and load it chunk-wise on demand.
- Strive for a high data load throughput.

5.2 Security Assessment Methodology

In the next three chapters, we describe three TEE-protected data structures. Namely, B^+-trees in Chapter 6, database dictionaries in Chapter 7, and file systems in Chapter 8. Using these data structures, we build outsourced systems for secure index searches, database queries, and group file sharing, respectively.

To assess the security of these systems, we first formulate an attacker model in each chapter. The attacker model states the capabilities of the attacker and our security assumptions about the TEE. Then, we follow different methods to assess the security of our systems: a formal security proof, a comparison to related work, and explicit security objectives. We describe the methods in the following and explain why we use them in each case.

Formal security proof. In Chapter 6, we present two constructions of TEE-protected B^+-tree, which we integrate into an outsourced system for index searches. The constructions are related to *symmetric searchable encryption* (SSE) and the security of the two constructions differs only slightly. Thus, we assess their security based on the three-step, formal-proof framework for SSE introduced by Curtmola *et al.* [144]: The first step of this framework is to formulate a leakage $\mathcal{L}^{enc}$, i.e., an upper bound of the information that an adversary can gather from the protocol. Secondly, one defines a $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and an $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ game for an adaptive adversary $\mathcal{A}$ and a polynomial time simulator $\mathcal{S}$. $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ is the actual protocol execution and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ uses $\mathcal{S}$ to simulate the real game by using only the formulated leakage $\mathcal{L}^{enc}$. An adaptive adversary can use information learned in previous protocol iterations for its queries. Third, a scheme is denoted CKA2-secure if one can show that $\mathcal{A}$ can distinguish the output of the games with negligible probability. If this is the case, $\mathcal{A}$ does not learn anything besides the leakage stated in the first step, because otherwise $\mathcal{A}$ could use this additional information to distinguish the games.

We extend this framework, because security models of SSE schemes only cover the transaction between the user and server in their leakage. TEE-based solutions, however, have an additional transaction between the server and the enclave, which is observable by $\mathcal{A}$. Therefore, we extend CKA2 security introducing $\mathcal{L}^{hw}$—a new type of leakage consisting of the inherent leakage of a

TEE and the inputs/outputs to/from the enclave. We denote the extended CKA2 security by CKA2-HW security.

In our security evaluation, we define the leakages $\mathcal{L}^{enc}$ and $\mathcal{L}^{hw}$ for both B$^+$-tree constructions and prove that $\mathcal{A}$ can distinguish the output of the $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and an $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.

Comparison to related work. In Chapter 7, we present nine TEE-protected database dictionary types, which we integrate in an outsourced, column-oriented, in-memory database. Each dictionary type has a different security level and the data owners can freely choose one dictionary type per database column according to their security requirements. We assess the security of six dictionary types by comparing them with security schemes or definitions known in literature. In more detail, we provide a security scheme or definition for six dictionary types with comparable security. By comparable security, we mean that this security scheme or definition leaks at least as much as the dictionary type, but it has the smallest leakage compared to others we found. Afterwards, we classify the relative security of the six dictionary types and integrate the three remaining dictionary types in this classification.

Explicit security objectives. In Chapter 8, we present a TEE-protected file system, which we integrate into an outsourced system for group file sharing. As many papers present such group file sharing systems, we analyze ten of them—six of them are purely cryptographically protected and four are TEE-supported. From this analysis, we derive a list of five important security objectives, with sub-objectives in some cases. We use these objectives to classify our system and the ten related systems.

5.3 Performance Assessment Methodology

To assess the performance of the three systems under a memory-limited, widely available *trusted execution environments* (TEEs), we implemented each system and performed experiments using Intel SGX. Each chapter has a dedicated section describing implementation details, e.g., the used programming languages, libraries, and Intel SGX SDK.

For the three systems, we measure the latency of the outsourced data processing, i.e., the time between sending a request and receiving a response, as we consider this the most important acceptance parameter for users. The measurements either exclude the network latency (Chapter 6 and 7) or include it (Chapter 8). All our systems only require one round of communication per request and there are good reasons for excluding and for including the network latency in this case. Excluding the network latency isolates the latency caused by the outsourced processing. Especially if the network latency is (severely) higher than the system's latency, it might hide the poor performance of the system. Furthermore, the network latency severely depends on the evaluation setup. Consequently, a comparison between systems is facilitated if the network latency is excluded. Including the network latency, however, gives more realistic latency results for the system's users as they are typically not co-located with the system.

In Chapter 6, we use a local Intel SGX machine for all latency measurements as no cloud provider offered machines with Intel SGX at the time of our evaluation. We provide latency measurements for both B$^+$-tree constructions and we compare the latency with the fastest related work supporting the same functionality. This related work is purely cryptography-based, because there was no TEE-based related work at the time of our evaluation.

In Chapter 7, we perform the measurements with one Intel SGX–enabled machine running at Microsoft Azure [172]. We provide detailed latency measurements for each dictionary type. Additionally, we compare these measurements with plaintext variants of each dictionary type

and with an open source, plaintext database implementation. Furthermore, we list the overhead induced by related, TEE-based solutions compared to a plaintext database.

In Chapter 8, we use two machines at Microsoft Azure distributed in the United States to provide a representative measurement for users of the group file sharing system. We show that the latency of our encrypted file sharing system is between two known file sharing systems that serve plaintext files. Furthermore, we use five performance objectives to classify our file sharing system, related purely cryptographically protected file sharing systems, and related TEE-protected file sharing systems.

In Chapter 7 and 8 we also evaluate the storage overhead of our encrypted systems compared to plaintext variants. This is important, because a system's performance can always be increased by using additional storage.

6
Protected B$^+$-trees: HardIDX

In this chapter, we describe the secure, outsourced, TEE-based processing of the *B$^+$-tree* data structure [173]. A B$^+$-tree is a balanced, n-ary search tree used to index *values* by *search keys*. Searches can be performed for single search keys or search key ranges. B$^+$-trees are frequently used for database indices in DBMSes [174], [175], for document indices [176], and for file systems [177], [178] to vastly improve the performance of search operations.

Using TEE-protected B$^+$-trees, we design *HardIDX*—a secure, efficient, outsourced system for index searches. Throughout this chapter, we only consider range searches, but HardIDX trivially supports equality searches. Furthermore, it can be adapted to other search operations, e.g., prefix, suffix, or substring search, and it can be deployed as a database index of an encrypted database.

We discuss two HardIDX constructions differing in the management of the B$^+$-tree. *Construction 1* loads the whole B$^+$-tree structure in the enclave memory and performs search queries thereafter. Due to Intel SGX's restricted EPC size (see Section 3.1.1), loading large trees into the EPC leads to paging, which introduces a major performance overhead. To mitigate this problem, *construction 2* only loads those data into the EPC that are currently needed to process a search query.

For HardIDX's security evaluation, we start with proofs for a passive attacker. Although the encrypted tree does not directly leak information about its content, it leaks access pattern information, which an attacker can observe, during query processing. For both constructions, we show the access pattern leakage with respect to Intel SGX's inherent side channels. Furthermore, we show that the leakage of both constructions is almost the same differing only by granularity. Based on the leakage discussion for a passive attacker, we present the security implications of an active attacker.

We implemented a prototype of HardIDX and use it for multiple performance evaluation experiments: a latency comparison of both constructions, a memory management impact of both constructions, and a comparison to the fastest related approach supporting range queries on encrypted data.

Index searches can also be implemented using cryptographic approaches that allow arbitrary computation on encrypted data, e.g., MPC [113], [179] or FHE. However, MPC and FHE schemes are still too inefficient for encrypted data search [19], [180]. Other cryptographic solutions are also applicable, e.g., PPE [10], [12], FE [157], or SE [140], [144], [148], [181], but performing efficient and secure *range* queries are commonly considered to be very challenging. CryptDB [182] resorts to OPE for this purpose which is susceptible to simple ciphertext-only attacks as shown by Naveed *et al.* [13]. Many SE schemes supporting range queries require search time linear in the number of database records [145]–[147]. Other papers use an index structure to achieve polylogarithmic search time [148], [149], [183]. Yet, the scheme by Lu [148] is inefficient, because it applies pairing-based cryptography, and it also leaks sensitive information about the plaintext, namely the order of the plaintexts. Demertzis *et al.* [149] and Faber *et al.* [183] present approaches with polylogarithmic search time utilizing only lightweight cryptography, that is, *pseudorandom function* (PRF) and *symmetric-key encryption* (SKE). Out of the many schemes presented by Demertzis *et al.* [149], the most secure approach, without

false positives and bearable storage cost achieves practical deployability. However, it still leaks a significant amount of sensitive information, e.g., the search pattern and the range size of each query, and it is orders of magnitude slower than HardIDX.

The contributions of this chapter are as follows:

- Two secure, efficient, outsourced, TEE-based index searches using protected B$^+$-trees. Both searches have a minimal enclave size, logarithmic complexity in the size of the index, and latency within a few milliseconds.
- Formal model and proof of the index searches showing that their security (leakage) is comparable to the best-known searchable encryption schemes.
- Security inspection of the index searches under an active attacker.
- Implementation and performance evaluation of the index searches.
- Evaluation of a performance bottleneck of Intel SGX based on two index searches with different memory-management strategies.

The remainder of this chapter is structured as follows: In Section 6.1, we provide a comprehensive introduction into B$^+$-trees; introduce HardIDX, the system in which we embed protected B$^+$-trees; and present the attacker model. In Section 6.2, we give an overview of related work. Afterwards, we focus on the design of two different HardIDX constructions under a passive attacker in Section 6.3. In Section 6.4, we present two extensions, which can be combined with both constructions. Namely, multi-user support and protection under an active attacker. In Section 6.5, we cover our HardIDX prototypes and explore implementation details relevant for security and performance. Based on the two HardIDX constructions and the implementation details, we provide in-depth security and performance evaluations in Section 6.6. Finally, Section 6.7 provides a summary of this chapter.

6.1 Design Considerations

In this section, we introduce the data structure under investigation, a B$^+$-tree; the system in which we embed TEE-protected B$^+$-trees, HardIDX; and the attacker model.

6.1.1 Data Structure: B$^+$-tree

A B$^+$-tree is a balanced, n-ary search tree, which indexes values by search keys. It can be used to search for a single search key, e.g., search for one staff id to find the corresponding database record (see Figure 6.1), and to perform range searches, e.g., search all staff ids between an upper and a lower limit to find the corresponding records. A range search can also be done for all search keys below or above a specified limit.

A B$^+$-tree $\boldsymbol{T}$ contains $|\boldsymbol{X}|$ nodes $\boldsymbol{X}$, and we differentiate three node types: the *root node* $T.root$, *inner nodes*, and *leaf nodes*. The root node $T.root$ becomes a leaf node if it is the only node in the tree, and an inner node otherwise. The *height* of a B$^+$-tree is defined by the length of the longest path from the root node to a leaf node and we denote it $T.h$. Additionally, we define $X{\rightarrow}parent_1$ as the *parent node* of X in $\boldsymbol{T}$, and $X{\rightarrow}parent_l$ specifies the node that is reached by moving l layers up in the tree $\boldsymbol{T}$ starting from node X.

Each node $X \in \boldsymbol{X}$ contains search keys $X.\boldsymbol{K}$ from a *search key domain* $\mathcal{D}$. The number of search keys is node-specific, and we denote it $|X.\boldsymbol{K}|$, i.e., $X.\boldsymbol{K} = (X.K_1, \ldots, X.K_{|X.\boldsymbol{K}|})$. The search keys are stored in a non-decreasing order, i.e., $X.K_1 \leq \ldots \leq X.K_{|X.\boldsymbol{K}|}$. Additionally, each node contains *pointers* $X.\boldsymbol{P}$. The number of pointers is $|X.\boldsymbol{K}| + 1$ and we denote it $|X.\boldsymbol{P}|$, i.e., $X.\boldsymbol{P} = (X.P_0, \ldots, X.P_{|X.\boldsymbol{K}|})$. The *branching factor* $T.b$ of a B$^+$-tree $\boldsymbol{T}$ defines the minimal and maximal number of pointers (and with it the minimal and maximal number of search keys) a node can have.

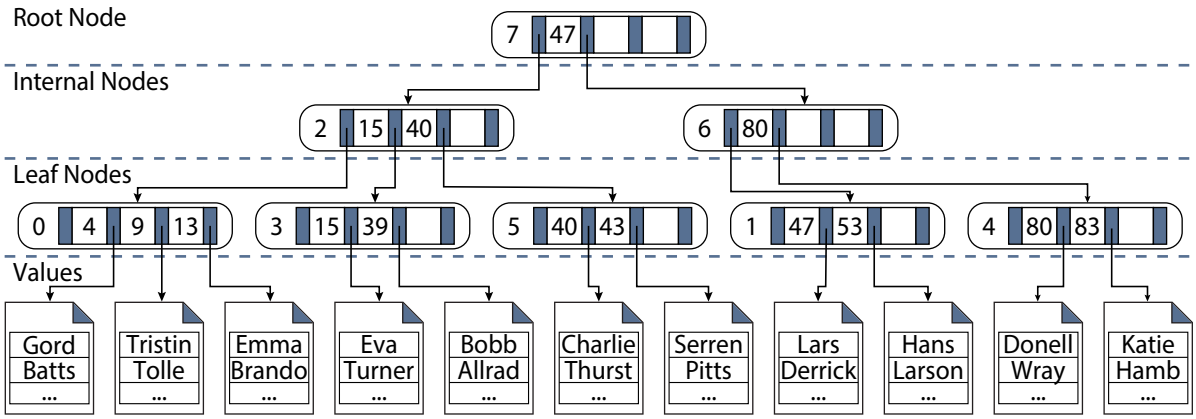

Figure 6.1: B$^+$-tree example: the search keys, which are unique staff ids, are used to index values, which are staff records. The random storage positions are illustrated on the left side of the nodes.

The following must apply at all times:

- Root node X is a leaf node: $1 \leq |X.\boldsymbol{P}| \leq T.b - 1$.
- Root node X is an inner node: $2 \leq |X.\boldsymbol{P}| \leq T.b$.
- Inner node X: $\lceil T.b/2 \rceil \leq |X.\boldsymbol{P}| \leq T.b$.
- Leaf node X: $\lceil T.b/2 \rceil \leq |X.\boldsymbol{P}| \leq T.b$.

Each inner node X stores search keys and pointers to *child nodes*. In more detail, every search key $X.K_i$ with $i \in [1, |X.\boldsymbol{K}|]$ has a corresponding pointer $X.P_i$ that points to a child node containing only search keys greater than or equal to $X.K_i$ and smaller than any other search key $X.K_j$ with $j \in [i+1, |X.\boldsymbol{K}|]$. $X.P_0$ points to a child node containing only search keys smaller than $X.K_1$. As a result, the $|X.\boldsymbol{K}|$ search keys separate the search key domain $\mathcal{D}$ into $(|X.\boldsymbol{K}| + 1) = |X.\boldsymbol{P}|$ subtrees that are reachable by $|X.\boldsymbol{P}|$ child pointers.

Each leaf node X store search keys and pointers to values[1]. In more detail, each pointer $X.P_i$ with $i \in [1, |X.\boldsymbol{K}|]$ points to the value indexed by $X.K_i$. $X.P_0$ is not used for leaf nodes. We denote all B$^+$-tree nodes without the values by *B$^+$-tree structure*.

The B$^+$-trees used in this chapter extend textbook B$^+$-trees with further attributes: each node $X \in \boldsymbol{X}$ has a unique id $X.id$ and stores if it is a leaf in $X.isLeaf$. A deviation from textbook B$^+$-trees is that we use unchained B$^+$-trees, i.e., the leaves are not connected. The reason is that a range query would directly leak the relationship among leaves if links are followed during a query. Linked leaves would increase the search performance, but it would severely deteriorate the security. Additionally, we denote the storage position (in the physical memory) of a node X by $X.pos$.

In contrast to other approaches using encrypted B$^+$-trees, we do not require to define the search key domain $\mathcal{D}$ in advance. It is not even necessary that the domain is a range of integers. Instead, $\mathcal{D}$ can be an arbitrary domain with a defined order relation and a defined minimal and a maximal element, i.e., $\forall\, X.K \in \mathcal{D} : -\infty < X.K < \infty$ with $-\infty$ and ∞ denoting the minimal and maximal domain values.

1 A main difference between B$^+$-trees and B-trees is that B$^+$-trees store pointers to values only in leaf nodes and B-trees store pointers to values in leaf and inner nodes.

6.1.2 System: HardIDX

Based on the B+-tree data structure described in the last section, we build HardIDX—a secure, efficient, outsourced index search using a TEE. Throughout this chapter, we consider a static B+-tree, i.e., the data stored in the tree is outsourced at one point in time. Afterwards, only search queries are possible and a data update would require a replacement of the whole tree.

Figure 6.2 presents the process flow and an overview of HardIDX involving two entities: a trusted user, who is the data owner, and an untrusted cloud server with a trusted (Intel SGX) enclave. Next, we describe HardIDX's setup and query phase.

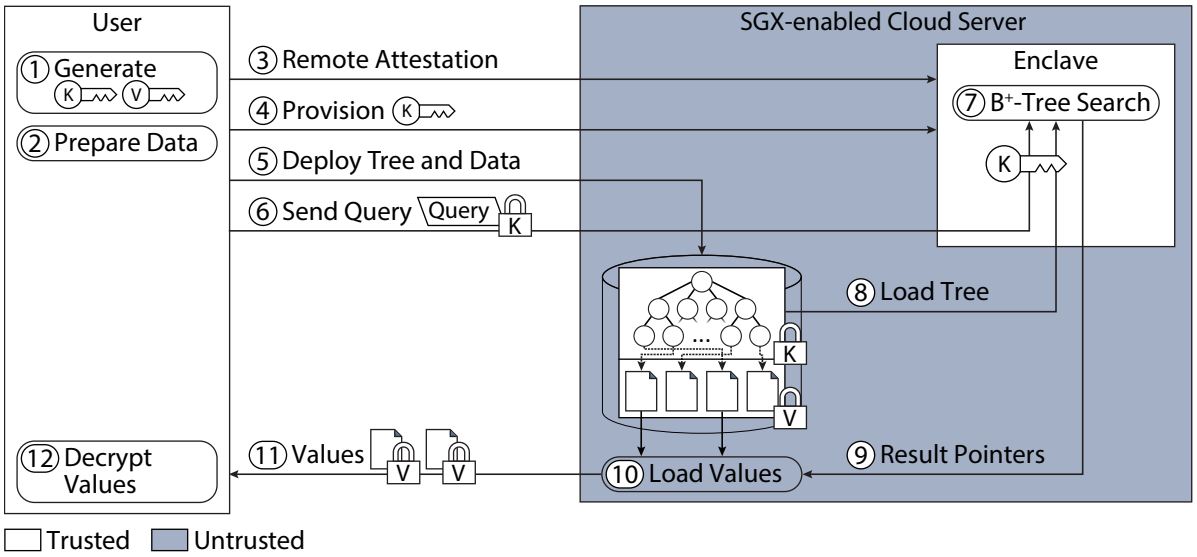

Figure 6.2: Overview and process flow of HardIDX. For visualization purposes, the tree nodes and values at the server are shown to be encrypted as a block. In reality, nodes and values are encrypted individually.

Setup phase. The setup phase is only executed once and consists of the following steps:

① The user generates two keys: SK^K and SK^V.
② The user prepares its values, which can be any data, e.g., rows in a relational database, values in a key-value store, or files. It stores all values in a pseudorandom order, augments the values with search keys, and inserts these search keys into a B+-tree. It then pseudorandomly shuffles the storage order of all nodes in the B+-tree. Finally, it uses an AE scheme (see Section 2.2.5) to encrypt all B+-tree nodes with SK^K and all values with SK^V.
③ The user uses Intel SGX's remote attestation feature to authenticate the enclave (see Section 3.1.3), and to establish an authenticated channel to the enclave.
④ The user uses the authenticated channel to provision SK^K into the enclave[2].
⑤ The user deploys the encrypted B+-tree nodes and encrypted values to the untrusted cloud.

Query phase. After the setup, the user can send range queries to the server. Note that it is straightforward to express equality queries, upper limit queries, and lower limit queries as range queries.

2 The enclave can use Intel SGX's sealing feature to securely store SK^K in untrusted memory (see Section 3.1.3). Sealing SK^K enables to reboot the enclave without repeating step ③ and ④.

⑥ The user encrypts the search range with AE under SK^K and sends the encrypted query to the enclave at the Intel SGX–enabled cloud server. The untrusted server cannot learn anything about the query, not even if the same query was sent before, because AE is a randomized encryption scheme.
⑦ The enclave has access to SK^K; thus, can decrypt the incoming query. It then starts the B^+-tree search
⑧ For this search, the enclave loads the required nodes from the untrusted storage into enclave memory and decrypts the nodes. In construction 1, the entire tree is loaded into the enclave and the search is performed afterwards. If the tree size exceeds the EPC memory, this leads to severe performance degradations. As a countermeasure, construction 2 loads only a subset of tree nodes into the enclave and loads further nodes on demand until the search is finished.
⑨ In both constructions the search algorithm eventually reaches a set of leaf nodes, which hold pointers to values matching the query. This list of pointers represents the search result. The enclave decrypts the pointers, shuffles the list, and passes the list to the untrusted part, which only learns the cardinality of the result set.
⑩ The untrusted part uses the pointers to fetch the encrypted values from untrusted storage.
⑪ The untrusted part sends the encrypted values to the user.
⑫ The user uses SK^V to decrypt the received values. Notably, SK^V never leaves the user; thus, plaintext values are never available on the server, not even inside the Intel SGX enclave.

6.1.3 Attacker Model

We assume that the cloud provider deploys HardIDX on a machine supporting a TEE with the capabilities listed in Table 3.1. The TEE does not protect against side channels, which could potentially be used by the attacker to extract sensitive information.

The attacker aims to learn any sensitive information about the B^+-tree, e.g., plaintext search keys, plaintext values, or the order relation between the indexed values. We assume the attacker has full control over all software on the system running HardIDX and thus can perform the following attacks:

1. Observe all interactions between the enclave and external resources. In particular, the attacker can observe the access pattern to B^+-tree nodes stored outside the enclave.
2. The attacker can use the page-fault side channel to observe data access inside the enclave at page granularity [32], [76]. Through this side channel, the attacker can observe access patterns on the B^+-tree stored inside the enclave.
3. The attacker can use a cache-timing side channel to learn about code paths or data access patterns inside the enclave [33], [77], [78].
4. The attacker can deviate from the defined protocol to gain additional sensitive information.

As we show in Section 3.2, Intel SGX is vulnerable to other (side-channel) attacks and multiple mitigations are known. In this chapter, we only consider the side channels mentioned before and consider the mitigation of other attacks an orthogonal problem. Nevertheless, HardIDX has a minimal enclave size; therefore, the presented mitigations should be straightforward to integrate. Hardware and *denial of service* (DoS) attacks are out of scope.

For ease of explanation, we first consider a passive attacker in our design section (Section 6.3) and explain protection against an active attacker as an extension in Section 6.4.2.

6.2 Related Work

HardIDX is related to TEE-based applications; software-only, encrypted databases; and searchable encryption.

6.2.1 TEE-based Applications

Besides the TEE-based applications that we discuss in Section 4.8, we consider the following applications related to HardIDX.

VC3 [184] adapts the MapReduce computing paradigm to Intel SGX. Mapper and Reducer entities are executed in individual enclaves and thus the data flow between them can leak sensitive information. While VC3 is tailored towards Intel SGX, they exclude information leakage from their adversary model. In contrast, we specifically focus on information leakage in the interaction of an enclave with other entities.

Ohrimenko *et al.* [185] present data-oblivious machine learning for Intel SGX. The authors adapt four machine learning algorithms to prevent side-channel exploitation. They transfer all secret-dependent data and code accesses to data-oblivious accesses using a library that provides a set of data-oblivious primitives. Access to external data, specifically input data, is addressed by randomizing the data and always accessing all data, i.e., their solution has a complexity of $O(n)$, even for tree searches. Following the same approach, HardIDX could trivially achieve data-oblivious access to the tree since the nodes of our tree are randomized as well. However, we would lose a main feature of our system: search time complexity of $O(\log n)$.

6.2.2 Software-only, Encrypted Databases

Some software-only, encrypted databases use PPE for efficient search, e.g., CryptDB [182] and Monomi [186]. PPE has a low deployment and runtime overhead due to the ability to use internal index structures of the database engine in the same way as on plaintext. However, the security of PPE schemes, such as DET, OPE, and ORE, is debatable (see Section 4.4.2). For instance, Naveed *et al.* [13], Grubbs *et al.* [14], and Lacharité *et al.* [15] present attacks recovering plaintext with a high success rate.

There have been a number of attempts to build indices for range queries based on DET. Bucketization groups ciphertexts on the server and filters results at the user [187]. Wang *et al.* [188] use distance-revealing encryption in order to build an r-tree. Li *et al.* [189] use prefix-preserving encryption in order to build a prefix tree for range searches. However, all of these approaches are susceptible to the attacks mentioned above.

Three further approaches for a secure DBMS allowing range queries have been published: Firstly, Pappas *et al.* [190] evaluate encrypted bloom filters using MPC. To achieve practical efficiency, the authors propose to split the server into two non-colluding parties. Our approach does not require any additional party. Secondly, Egorov *et al.* [191] present ZeroDB, a database system that enables a user to perform equality and range searches with the help of B^+-trees. ZeroDB is an interactive protocol requiring many rounds and thus is unsuitable for network sensitive cloud computing. Thirdly, Poddar *et al.* [192] present Arx. This system uses a binary tree with garbled circuits at every node. The garbled circuits evaluate the tree traversal decisions in a protected manner, but any traversal destroys the visited garbled circuits. Therefore, the user has to provide new circuits in an additional round or with the next query. This interactive reparation step severely reduces the usefulness in a highly concurrent cloud scenario.

6.2.3 Searchable Encryption (SE)

In Section 4.5, we provide a detailed introduction into SE schemes with various processing capabilities and security definitions. Here, we compare HardIDX only with SE schemes supporting range queries, because HardIDX also supports this query type. Table 6.1 shows a summary of the comparison results.

Scheme	Search time	Query size	Storage size	Search pattern leakage	Order leakage
Boneh *et al.* [145]	$O(n\,\lvert\mathcal{D}\rvert\,)$	$O(\lvert\mathcal{D}\rvert\,)$	$O(n\,\lvert\mathcal{D}\rvert\,)$	●	○
Shi *et al.* [147]	$O(n\log\lvert\mathcal{D}\rvert\,)$	$O(\log\lvert\mathcal{D}\rvert\,)$	$O(n\log\lvert\mathcal{D}\rvert\,)$	●	○
Shen *et al.* [146]	$O(n\log\lvert\mathcal{D}\rvert\,)$	$O(\log\lvert\mathcal{D}\rvert\,)$	$O(n\log\lvert\mathcal{D}\rvert\,)$	○	○
Lu [148]	$O(\log n\log\lvert\mathcal{D}\rvert\,)$	$O(\log\lvert\mathcal{D}\rvert\,)$	$O(n\log\lvert\mathcal{D}\rvert\,)$	○	●
Demertzis *et al.* [149] Faber *et al.* [183]	$O(\log R)$	$O(\log R)$	$O(n\log\lvert\mathcal{D}\rvert\,)$	●	○
HardIDX	$O(\log n)$	$O(1)$	$O(n)$	○	○

Table 6.1: Comparison of range-searchable encryption schemes. n is the number of indexed search keys, $|\mathcal{D}|$ is the size of the plaintext domain, and R is the query range size. The symbols represent that a leakage is present (●) or not present (○).

Boneh *et al.* [145] present the first range-searchable scheme, which encrypts every entry linear in the size of the plaintext domain. The scheme from Shi *et al.* [147] only requires logarithmic storage per entry in the domain, but their scheme additionally leaks the plaintext of matching ciphertexts. The construction is based on inner-product predicate encryption, which Shen *et al.* [146] make fully secure. All of these schemes have linear search time.

Lu [148] builds the range-searchable encryption from Shen *et al.* into an index, thereby enabling polylogarithmic search time. However, his encrypted inverted index tree reveals the order of the plaintexts and is hence only as secure as OPE. Wang *et al.* [193] propose a multidimensional extension of Lu's scheme, but the extension suffers from the same order leakage problem. We implemented the range-searchable encryption proposed by Lu and ascertained that the approach requires several seconds or minutes for a single range search, even with a security parameter much weaker than HardIDX's security parameter. Hence, we not only improve asymptotic search time, but also reduce the constants enabling range-searchable encryption on much larger datasets.

Cash *et al.* [194] introduce a protocol called OXT that allows evaluation of boolean queries on encrypted data. Faber *et al.* [183] extend OXT to support range queries but either leak additional information on the queried range or the result set contains false positives. Demertzis *et al.* [149] present several approaches supporting secure range queries. They evaluate the security and performance of their approaches based on the OXT protocol. The scheme that is most comparable to HardIDX, Logarithmic-URC, is quite equal to the range query approach without false positives from Faber *et al.* and also exhibits additional leakage. We provide an experimental comparison in Section 6.6.2.

There is no known searchable encryption scheme for ranges that has polylogarithmic search time and leaks only the access pattern—until HardIDX. *Oblivious RAM* (ORAM) can be used to hide the access pattern, but Naveed [195] shows that the combination of the two is not straightforward. Special ORAM techniques, such as TWORAM [196], are needed.

6.3 Design

The focus of this section is on two HardIDX constructions, which enable users to search for single search keys and search key ranges on data that is outsourced to an untrusted cloud provider. First, we introduce the concept of a hardware secured B$^+$-tree (HSBT) and provide corresponding correctness and security definitions. An HSBT splits the execution of a B$^+$-tree search into trusted client-side, untrusted server-side, and trusted server-side algorithms. Afterwards, we present the two HardIDX constructions, which instantiate an HSBT, achieve logarithmic search complexity, and use Intel SGX to protect the confidentiality and integrity of the outsourced data under a passive attacker. We present protection strategies against an active attacker as a HardIDX extension in Section 6.4.2.

6.3.1 Hardware Secured B$^+$-tree

Based on the B$^+$-tree introduction in Section 6.1.1, we define the notion of a hardware secured B$^+$-tree (HSBT) as follows. We assume that a B$^+$-tree stores n search keys $\boldsymbol{K} = (K_1, \dots K_n)$ and their corresponding values $\boldsymbol{V} = (V_1, \dots, V_n)$. $\boldsymbol{S}$ denotes the set of key-value pairs, i.e., $\boldsymbol{S} = ((K_1, V_1), \dots, (K_n, V_n))$.

Definition 30 (HSBT)**.** *An* HSBT *scheme is a tuple of six PPT algorithms* (HSBT_Setup, HSBT_Enc, HSBT_Tok, HSBT_Dec, HSBT_SearchRange, HSBT_SearchRange_Trusted)*.*

Algorithms executed at the user:

$SK \leftarrow$ HSBT_Setup(1^λ): *Take a security parameter λ as input and output a secret key SK.*

$\gamma \leftarrow$ HSBT_Enc$(SK, \boldsymbol{S})$: *Take a secret key SK and key-value pairs $\boldsymbol{S}$ as input. Output an encrypted B^+-tree γ.*

$\tau \leftarrow$ HSBT_Tok$(SK, \boldsymbol{R})$: *Take a secret key SK and a range $\boldsymbol{R} = [R^s, R^e]$ as input. Output an encrypted search token τ.*

$\boldsymbol{V'} \leftarrow$ HSBT_Dec$(SK, \boldsymbol{C'})$: *Take a secret key SK and a set of ciphertexts $\boldsymbol{C'}$ as input. Decrypt the ciphertexts and output plaintext values $\boldsymbol{V'}$.*

Executed at the server on untrusted hardware:

$\boldsymbol{C'} \leftarrow$ HSBT_SearchRange(τ, γ): *Take an encrypted search token τ and optionally an encrypted tree γ as input and call the enclave function* HSBT_SearchRange_Trusted*. Output a set of encrypted values $\boldsymbol{C'}$.*

Executed at the server in an enclave:

$\boldsymbol{P} \leftarrow$ HSBT_SearchRange_Trusted$(\tau, \boldsymbol{X})$: *Take an encrypted search token τ and optionally nodes $\boldsymbol{X}$ as input. Output a set of pointers $\boldsymbol{P}$.*

In the following definitions, we assume a passive attacker. The security implications of an active attacker are presented in Section 6.6.

Definition 31 (HSBT correctness)**.** *Let $\mathcal{G}$ denote an* HSBT *scheme consisting of the six algorithms described in Definition 30. Given a passive attacker, we say that $\mathcal{G}$ is correct if for all $\lambda \in \mathbb{N}$, for all SK output by* HSBT_Setup(1^λ)*, for all key-value pairs $\boldsymbol{S}$ used by* HSBT_Enc$(SK, \boldsymbol{S})$ *to output γ, for all $\boldsymbol{R}$ used by* HSBT_Tok$(SK, \boldsymbol{R})$ *to output τ, for all $\boldsymbol{C'}$ output by* HSBT_SearchRange(τ, γ)*, the values $\boldsymbol{V'}$ output by* HSBT_Dec$(SK, \boldsymbol{C'})$ *are all values in $\boldsymbol{S}$ for which the corresponding search keys fall into $\boldsymbol{R}$, i.e., $\boldsymbol{V'} = \{V_i \,|\, (K_i, V_i) \in \boldsymbol{S} \wedge K_i \in [R^s, R^e] = \boldsymbol{R}\}$.*

We define the security of a HSBT based on the extended framework for SSE security proofs introduced in Section 5.2:

Definition 32 (HSBT CKA2-HW security)**.** *Let $\mathcal{G}$ denote an* HSBT *scheme consisting of the six algorithms described in Definition 30. Consider the probabilistic experiments $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$, whereas $\mathcal{A}$ is a stateful passive adversary and $\mathcal{S}$ is a stateful simulator that gets the leakage functions $\mathcal{L}^{enc}$ and $\mathcal{L}^{hw}$.*

$\boldsymbol{Real}_{\mathcal{A}}(\lambda)$: *The challenger runs* $\mathsf{HSBT\_Setup}(1^{\lambda})$ *to generate a secret key SK. $\mathcal{A}$ outputs key-value pairs $\boldsymbol{S} = ((K_1, V_1), \ldots, (K_n, V_n))$. The challenger calculates $\gamma \leftarrow \mathsf{HSBT\_Enc}(SK, \boldsymbol{S})$ and passes γ to $\mathcal{A}$. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries for arbitrary ranges $\boldsymbol{R}$. For each query, the challenger calculates $\tau \leftarrow \mathsf{HSBT\_Tok}(SK, \boldsymbol{R})$ and returns the encrypted search token τ. $\mathcal{A}$ can use γ and the returned tokens at any time to make queries to the enclave. The enclave returns a set of pointers $\boldsymbol{P}$. Finally, $\mathcal{A}$ returns a bit b that is the output of the experiment.*

$\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$: *The adversary $\mathcal{A}$ outputs key-value pairs $\boldsymbol{S} = ((K_1, V_1), \ldots, (K_n, V_n))$. Using $\mathcal{L}^{enc}$, $\mathcal{S}$ creates γ and passes it to $\mathcal{A}$. Afterwards, $\mathcal{A}$ makes a polynomial number of adaptive queries for arbitrary ranges $\boldsymbol{R}$. For each query, the simulator $\mathcal{S}$ creates an encrypted search token τ and passes it to $\mathcal{A}$. The adversary $\mathcal{A}$ can use γ and the returned tokens at any time to make queries to $\mathcal{S}$ (that simulates the enclave). $\mathcal{S}$ is given $\mathcal{L}^{hw}$ and returns a set of pointers $\boldsymbol{P}$. Finally, $\mathcal{A}$ returns a bit b that is the output of the experiment.*

$\mathcal{G}$ is CKA2-HW secure against adaptive chosen-keyword attacks if for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that

$$|\mathsf{Pr}[\boldsymbol{Real}_{\mathcal{A}}(\lambda) = 1] - \mathsf{Pr}[\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

6.3.2 Construction 1

In this section, we describe our first correct (according to Definition 31) and secure (according to Definition 32) HardIDX construction. The guiding idea of construction 1 is that the entire B^+-tree structure should be stored and processed inside the enclave. We start with a brief repetition of HardIDX's setup phase (see Section 6.1.2) enriched with details about construction 1. Then, we describe how construction 1 is an initialization of an HSBT scheme and present its correctness and security. Finally, we discuss the problems of construction 1.

Setup phase. The cloud provider deploys the server-side algorithms, which are described later, to an Intel SGX–enabled cloud server. The user generates SK^K and SK^V, constructs the B^+-tree, encrypts the B^+-tree structure with an AE scheme under SK^K, and encrypts the values with an AE scheme under SK^V. Then, the user uses Intel SGX's remote attestation protocol (see Section 3.1.3) to authenticate the enclave used for the B^+-tree search, establishes an authenticated channel to the enclave, and uses this channel to deploy SK^K inside the enclave. As a result, SK^K is only known by the user and the enclave, and SK^V never leaves the user. The enclave can use sealing (see Section 3.1.4) to securely store SK^K in untrusted memory, which would support enclave reboots without repeating remote attestation and key transfer.

Next, the user sends the encrypted B^+-tree structure and the encrypted values to an untrusted storage region at the cloud server. The enclave loads the B^+-tree structure into the enclave memory (see Section 3.1.1), decrypts the nodes with SK^K, and keeps the plaintext nodes in the enclave memory. Remember that the nodes are still protected, because Intel SGX guarantees confidentiality and integrity for all data inside the enclave memory. The encrypted values, which can be huge, remain in untrusted memory. Consequently, less data resides in the enclave memory, which reduces the necessity of slow EPC paging (see Section 3.1.1). Note that this has no negative security implications as the values are encrypted with an AE scheme and the

values are stored in a randomized order. After this setup, the enclave is ready to receive search queries from the user.

HSBT scheme. Construction 1 is an HSBT scheme denoted by HSBT1 consisting of six PPT algorithms (HSBT1_Setup, HSBT1_Enc, HSBT1_Tok, HSBT1_Dec, HSBT1_SearchRange, HSBT1_SearchRange_Trusted). The scheme uses a pseudorandom permutation $\mathsf{PRP} : \{0,1\}^\lambda \times \{0,1\}^{|\boldsymbol{X}|} \rightarrow \{0,1\}^{|\boldsymbol{X}|}$.

Algorithms executed at the user:

$\boldsymbol{SK} \leftarrow \mathsf{HSBT1\_Setup}(1^\lambda)$: Take the security parameter λ as input to execute $\mathsf{AE\_Gen}(1^\lambda)$ two times generating SK^V and SK^K. Output $\boldsymbol{SK} = (SK^K, SK^V)$. Keep SK^V and SK^K secret at the user. Share SK^K with the enclave at the cloud server using the authenticated channel established during remote attestation.

$\gamma \leftarrow \mathsf{HSBT1\_Enc}(\boldsymbol{SK}, \boldsymbol{S})$: Take the key $\boldsymbol{SK}$ and the key-value pairs $\boldsymbol{S} = ((K_1, V_1), \ldots, (K_n, V_n))$ as input. Store all values $\boldsymbol{V} = (V_1, \ldots, V_n)$ in a random order. Then, insert all search keys into a B+-tree using a textbook B+-tree insertion with five variations:

1. Give every newly created node X an id according to the creation order, i.e., $X_0.id = 0$ for the first node X_0, $X_1.id = 1$ for the second node X_1 et cetera.
2. After all search keys are inserted, fill up all nodes with search keys and values. More specifically, fill a node X that contains $|X.\boldsymbol{K}|$ search keys from the domain $\mathcal{D}$ with $(T.b - 1 - |X.\boldsymbol{K}|)$ times the search key ∞ and $(T.b - |X.\boldsymbol{K}|)$ dummy pointers.
3. Pad all search keys and pointers to a fixed bit-length.
4. Use the node ids to store each node at a pseudorandom position, i.e., a node X with id $X.id$ is stored at position $X.pos = \mathsf{PRP}(SK^K, X.id)$.
5. Use $\mathsf{AE\_Enc}(SK^V, \cdot)$ to encrypt each value $V \in \boldsymbol{V}$ and use $\mathsf{AE\_Enc}(SK^K, \cdot)$ to encrypt each node $X \in \boldsymbol{X}$.

Return the encrypted tree γ, which consists of the encrypted values and nodes.

The described variations lead to an encrypted B+-tree in which every node occupies the same storage space, the order of the nodes is random, and the order of the values is random.

$\tau \leftarrow \mathsf{HSBT1\_Tok}(SK^K, \boldsymbol{R})$: Take SK^K and a range $\boldsymbol{R} = [R^s, R^e]$ as input, calculate $\tau \leftarrow \mathsf{AE\_Enc}(SK^K, R^s || R^e)$, and output τ. Queries for all elements below R^e or all elements above R^s can be created by using $R^s = -\infty$ or $R^e = \infty$, respectively.

$\boldsymbol{V'} \leftarrow \mathsf{HSBT1\_Dec}(SK^V, \boldsymbol{C'})$: Take SK^V and the encrypted values $\boldsymbol{C'} = (C_0, \ldots, C_j)$ as input, decrypt all values, i.e., $\boldsymbol{V'} = (\mathsf{AE\_Dec}(SK^V, C_0), \ldots, \mathsf{AE\_Dec}(SK^V, C_j))$, and output $\boldsymbol{V'}$.

Executed at the server on untrusted hardware:

$\boldsymbol{C'} \leftarrow \mathsf{HSBT1\_SearchRange}(\tau)$: Take an encrypted search token τ as input, pass τ to HSBT1_SearchRange_Trusted, and return the result values $\boldsymbol{C'} = (C_0, \ldots, C_j)$ by dereferencing the received pointers. See Algorithm 1 for details.

Algorithm 1 HSBT1_SearchRange(τ)

1: $\boldsymbol{P}^{res} = \mathsf{HSBT1\_SearchRange\_Trusted}(\tau)$
2: **return** $*P_0^{res}, *P_1^{res}, \ldots$ ▷ Return dereferenced pointers

Executed at the server in an enclave:

$\boldsymbol{P} \leftarrow \mathsf{HSBT1\_SearchRange\_Trusted}(\tau)$: Take the encrypted search token τ as input, decrypt the token, perform a B+-tree traversal, and return the pointers corresponding to values falling in the queried range in a random order. See Algorithm 2 for details.

During the B$^+$-tree traversal, the algorithm accesses all search keys in every accessed node to mitigate secret-dependent decisions. Furthermore, it shuffles the currently processed nodes $\boldsymbol{X}$ in every round to hide the order. Note that the enclave does not have to decrypt the nodes, because the user's encryption was removed in the setup phase. The nodes reside in the enclave memory and therefore the CPU has plaintext access to them.

Algorithm 2 HSBT1_SearchRange_Trusted(τ)

```
1: τ^plain = AE_Dec(SK^K, τ)
2: parse τ^plain as (R^s, R^e)
3: P = ∅                                              ▷ Result pointers
4: X = (T.root)                                       ▷ Currently processed nodes
5: while X ≠ ∅ do
6:     P^tmp = ∅                                      ▷ Temporary pointers
7:     X = X.Pop()
8:     if not X.isLeaf && R^s < X.K_1 then
9:         P^tmp.Append(X.P_0)
10:    for i = 1; i < T.b − 1; i++ do
11:        If (X.K_i ≤ R^s < X.K_{i+1}) || (X.K_i ≤ R^e < X.K_{i+1}) ||
               (R^s ≤ X.K_i && X.K_{i+1} ≤ R^e) then
12:            P^tmp.Append(X.P_i)
13:    if X.K_{b−1} ≤ R^e then
14:        P^tmp.Append(X.P_{T.b−1})
15:    for P in P^tmp do
16:        if X.isLeaf then
17:            P.Append(P)
18:        else
19:            X.Append(*P)
20:    X = random permutation of X
21: P = random permutation of P
22: return P
```

Correctness and security. Construction 1 is correct according to Definition 31, because it basically performs a textbook B$^+$-tree traversal inside the enclave. The only difference is that the accessed nodes and returned pointers are randomized. Furthermore, the encryption (at the user) and the decryption (inside the enclave) are based on a correct AE scheme.

The following theorem states the security of construction 1 and it is proven in our security evaluation (Section 6.6.1):

Theorem 1 (HSBT1 security)**.** *The hardware secured B$^+$-tree construction* HSBT1 *is CKA2-HW secure according to Definition 32.*

Problems. Construction 1 suffers from the substantial problem mentioned before: the main memory region reserved for all enclaves running on an Intel SGX–enabled CPU—the EPC—is limited to about 96 MB. If the enclaves use more space, paging occurs. According to our experiments, at least 50,000,000 values are possible with construction 1. However, if a DBMS should be protected by Intel SGX, a B$^+$-tree is only a small part of the whole system. Other components would occupy large regions of the EPC and thus limit the available space for the encrypted B$^+$-tree structure. For that reason, we present a second construction in the next section that mitigates the EPC restriction issue.

6.3.3 Construction 2

In this section, we describe our second correct (according to Definition 31) and secure (according to Definition 32) HardIDX construction. Instead of loading all nodes into the enclave, the main idea is to load the nodes required to traverse the tree step by step. The challenge is to optimize the communication bottleneck between the untrusted part and the enclave. We performed extensive benchmarking and algorithm engineering in order to identify and minimize time-consuming tasks, such as switches between the untrusted part and the enclave. The decisive advantage of our second construction is that the required memory space inside the enclave is $O(1)$ for a tree of arbitrary size. The trade-off is that all nodes are stored encrypted inside untrusted main memory or on the untrusted disk; therefore, the nodes have to be decrypted by the enclave. Compared to construction 1, this leads to a slightly larger leakage, namely a finer-granular access pattern on node level instead of page level. Details are described by a formal model and proof in the security evaluation section (Section 6.6.1).

In the following, we describe the setup phase of construction 2, show how construction 2 is an initialization of an HSBT scheme; and present the correctness and security of construction 2.

Setup phase. As in construction 1, the user generates SK^K and SK^V, constructs the B^+-tree, encrypts the B^+-tree using the two keys, performs remote attestation, deploys SK^K inside the enclave using an authenticated channel, and sends the encrypted B^+-tree to the cloud server. In contrast to construction 1, the search algorithm in the enclave does not load the complete B^+-tree structure into the enclave memory. Instead, it only reserves a fixed space, which we denote `reservedSpace`, for on the fly processing. Let s be the block size of the used encryption scheme and o the number of blocks required by each node. Then, the search algorithm loads at most $\texttt{maxAmount} = \lfloor \texttt{reservedSpace}/(s \cdot o) \rfloor$ nodes into the enclave memory at the same time.

HSBT scheme. Construction 2 is an HSBT scheme denoted by HSBT2 consisting of six PPT algorithms (HSBT2_Setup, HSBT2_Enc, HSBT2_Tok, HSBT2_Dec, HSBT2_SearchRange, HSBT2_SearchRange_Trusted). The scheme uses a pseudorandom permutation $\mathsf{PRP} : \{0,1\}^\lambda \times \{0,1\}^{|\boldsymbol{X}|} \rightarrow \{0,1\}^{|\boldsymbol{X}|}$.

All algorithms but HSBT2_SearchRange and HSBT2_SearchRange_Trusted exactly match the corresponding algorithms of HSBT1. Thus, we only describe the two modified algorithms in the following. Compared to the HSBT1 versions, HSBT2_SearchRange additionally receives the encrypted tree γ and HSBT2_SearchRange_Trusted receives a tuple of nodes $\boldsymbol{X}$.

Executed at the server on untrusted hardware:

$\boldsymbol{C'} \leftarrow \mathsf{HSBT2\_SearchRange}(\tau, \gamma)$: Take an encrypted search token τ and the encrypted tree γ as input. Initially, pass only γ's root node $\gamma.root$ to the enclave and receive pointers to nodes that should be traversed next. Pass nodes until no further nodes are requested. Then, output $\boldsymbol{C'}$ by dereferencing all pointers to the values. See Algorithm 3 for details.

A trivial solution is to pass one node after another to the enclave, but each context switch from the untrusted part to the enclave causes a substantial performance overhead. Instead, the algorithm optimizes the number of context switches transferring as many nodes as currently in the queue, but not more than `maxAmount`.

Executed at the server in an enclave:

$\boldsymbol{P} \leftarrow \mathsf{HSBT2\_SearchRange\_Trusted}(\tau, \boldsymbol{X})$: Take an encrypted search token τ and nodes $\boldsymbol{X}$ as input. Decrypt τ and $\boldsymbol{X}$ using SK^K, which was deployed to the enclave during the setup phase. Then, search in all nodes for search keys falling in the query range. Finally, return the pointers corresponding to the found search keys in a random order. See Algorithm 4 for details.

During the search for search keys falling in the query range, the algorithm accesses all search keys to mitigate secret-dependent decisions.

Algorithm 3 HSBT2_SearchRange(τ, γ)

1: $\boldsymbol{P}^{res} = \emptyset$ ▷ Result pointers
2: $\boldsymbol{X} = (\gamma.root)$ ▷ Currently processed nodes
3: **while** $\boldsymbol{X} \neq \emptyset$ **do**
4: **for** i=0; i < $|\boldsymbol{X}|$ && i < maxAmount; i++ **do**
5: $\boldsymbol{X}^{tmp}$.Append($\boldsymbol{X}$.Pop())
6: $\boldsymbol{P}^{tmp}$ = HSBT2_SearchRange_Trusted$(\tau, \boldsymbol{X}^{tmp})$
7: **for** ($isLeaf$, P^{tmp}) in $\boldsymbol{P}^{tmp}$ **do**
8: **if** $isLeaf$ **then**
9: $\boldsymbol{P}^{res}$.Append(P^{tmp})
10: **else**
11: $\boldsymbol{X}$.Append($*P^{tmp}$)
12: **return** $*P_0^{res}, *P_1^{res}, \dots$ ▷ Return dereferenced pointers

Algorithm 4 HSBT2_SearchRange_Trusted$(\tau, \boldsymbol{X})$

1: $\boldsymbol{P} = \emptyset$ ▷ Result pointers
2: τ^{plain} = AE_Dec(SK^K, τ)
3: parse τ^{plain} as (R^s, R^e)
4: $\boldsymbol{X}^{plain} = (\text{AE\_Dec}(SK^K, X_0), \text{AE\_Dec}(SK^K, X_1), \dots)$ ▷ Currently processed nodes
5: **for** X in $\boldsymbol{X}^{plain}$ **do**
6: **if** not $X.isLeaf$ and $R^s < X.k_1$ **then**
7: $\boldsymbol{P}$.Append($X.P_0$)
8: **for** $i = 1$, $i < T.b - 1$, i++ **do**
9: **if** $(X.k_i \leq R^s < X.k_{i+1}) \,||\, (X.k_i \leq R^e < X.k_{i+1}) \,||\, (R^s \leq X.k_i \;\&\&\; X.k_{i+1} \leq R^e)$ **then**
10: $\boldsymbol{P}$.Append($X.P_i$)
11: **if** $R^e \geq X.k_{b-1}$ **then**
12: $\boldsymbol{P}$.Append($X.P_{b-1}$)
13: $\boldsymbol{P}$ = random permutation of $\boldsymbol{P}$
14: **return** $((*P_0^{res}.isLeaf, P_0^{res}), (*P_1^{res}.isLeaf, P_1^{res}), \dots)$

Correctness and security. Construction 2 is correct according to Definition 31, because it is based on a textbook B$^+$-tree traversal. The difference to the textbook algorithm is that the nodes are loaded into the enclave one after another and that each node is encrypted. These changes do not influence the correctness, because each node remains accessible to the enclave, as a passive attacker provides the requested nodes. Furthermore, the encryption (at the user) and the decryption (inside the enclave) are based on a correct AE scheme.

The following theorem states the security of construction 2 and it is proven in our security evaluation (Section 6.6.1):

Theorem 2 (HSBT2 security)**.** *The hardware secured B^+-tree construction* HSBT2 *is CKA2-HW secure according to Definition 32.*

6.4 Extensions

In this section, we present two extensions that can be used for both HardIDX constructions: multiple user support and protection under an active attacker. The extensions are independent, and it is straightforward to combine them.

6.4.1 Multiple User Support

So far, we considered a setup comprising one user, who is also the data owner (i.e., the user that prepared and uploaded the data). Considering multiple users, we differentiate between the data owner and users (see scenario presented in Figure 1.1b). The data owner performs all setup steps described before. If all users have access to the key SK^K to create query tokens and SK^V to decrypt the result, HardIDX supports multiple users without any change. The reason is that concurrent tree traversals do not influence each other.

With three protocol modifications, it is possible to hide the search pattern of users from each other:

1. The data owner encrypts all nodes with SK^K, performs remote attestation with the enclave, and shares SK^K with the enclave, but not with the users.
2. Each user performs remote attestation with the enclave and deploys its own key SK^U at the enclave.
3. Each user uses its key SK^U for query encryption and attaches a user identifier to the query. The enclave uses the identifier to decrypt the query with the corresponding key SK^U, performs a regular search, and re-encrypts the result with SK^U.

6.4.2 Protection Under an Active Attacker

In Section 6.3.2 and Section 6.3.3, we present constructions 1 and 2. Theorem 1 and 2 state that the constructions are secure under a passive attacker, and proofs for these theorems follow in our security evaluation (Section 6.6.1). The main goal of HardIDX, however, is an outsourced index search at untrusted cloud providers. In this case, it is important to consider an active attacker, i.e., an attacker that tries to thwart the correctness and tries to gain additional sensitive information by not following the defined protocol.

The security evaluation provides an in-depth discussion about possible attack vectors of an active attacker. For this section, only the result of this discussion is important, which states that an activate attacker can only do the following protocol deviations: (1) do not pass the root node first to the enclave, (2) do not pass all requested nodes to the enclave, and (3) do not pass all results to the user.

In the following, we focus on modifications for construction 2 that prevent these deviations. It is straightforward to adapt them to construction 1. The modifications involve the B+-tree creation at the user, the data transferred via the authenticated channel, the algorithm HSBT2_SearchRange, the algorithm HSBT2_SearchRange_Trusted, and result processing at the user. On a high level, deviation (1) is mitigated assigning a nonce to each query, and deviations (2) and (3) are mitigated managing multiple set hashes (see Section 2.2.7) for each nonce. The modifications only require $O(1)$ additional enclave memory per query.

- **Changes to B+-tree creation.** Each leaf node X stores the plaintext hash $X.hash_i = \mathsf{H}(V_i)$ of each linked value V_i next to the corresponding pointer $X.P_i$. Each non-leaf node X stores the id of each child $X.chId_i$ next to the corresponding pointer $X.P_i$.
- **Changes to data transferred via the authenticated channel.** Besides the key SK^K, the user transfers the id of the root node $X^{root}.id$ to the enclave.
- **Changes to HSBT2_SearchRange and HSBT2_SearchRange_Trusted.** For clarity, we abbreviate HSBT2_SearchRange and HSBT2_SearchRange_Trusted with Untrusted and Trusted, in the following description. W.l.o.g., we assume that Untrusted sends one node X at a time to Trusted.

  Trusted decrypts X and checks if $X.id$ matches $X^{root}.id$. If this is the case, the algorithm creates a new nonce, and stores the nonce in a list of known nonces. Otherwise, Trusted

expects to receive a nonce from Untrusted that is in the list of known nonces. Trusted aborts if it neither receives the root node nor a known nonce, which prevents deviation (1).

If a new nonce was created, Trusted initializes four attributes belonging to the nonce: number of expected nodes $expectedNodesNumber = 1$, a set hash for expected nodes $expectedNodesHash \leftarrow \mathsf{SH}(\emptyset)$, a set hash for received nodes $receivedNodesHash \leftarrow \mathsf{SH}(X.id)$, and a set hash for result values $resultValuesHash \leftarrow \mathsf{SH}(\emptyset)$. Otherwise, Trusted loads those attributes.

Afterwards, Trusted reduces $expectedNodesAmount$ by one, calculates $receivedNodesHash \leftarrow \mathsf{SH\_NI\_Add}(receivedNodesHash, \mathsf{SH}(X.id))$, and searches for search keys falling in the search range as described in the unchanged algorithm. If X is a non-leaf node, Trusted calculates $expectedNodesHash \leftarrow \mathsf{SH\_NI\_Add}(expectedNodesHash, \mathsf{SH}(X.chId_i))$ for each $X.P_i$ pointing to a child node falling into the search range. If X is a leaf node, Trusted calculates $resultValuesHash \leftarrow \mathsf{SH\_NI\_Add}(resultValuesHash, \mathsf{SH}(X.hash_i))$ for each hash $X.hash_i$ corresponding to a search key falling into the search range. Trusted increases $expectedNodesNumber$ according to the expected number of nodes. Finally, it passes a shuffled list of pointers and the nonce to Untrusted.

At one point, Trusted reduces $expectedNodesNumber$ to 0 with an incoming node and the search does not lead to more expected nodes. Then, Trusted calculates $b \leftarrow \mathsf{SH\_Comp}(expectedNodesHash, receivedNodesHash)$. If $b = 0$, Trusted did not receive the correct nodes from Untrusted, aborts and deletes the nonce and the set hashes. Thus, Trusted guarantees that it traversed all nodes that might contain an eligible result and thereby protects against deviation (2). If the hashes match, Trusted calculates $resultValuesHash \leftarrow \mathsf{SH\_NI\_Add}(resultValuesHash, \mathsf{SH}(X.hash_i))$ for each result found in the last search round, calculates $t \leftarrow \mathsf{MAC\_Tag}(SK^K, resultValuesHash)$, and returns t together with the pointers. Untrusted adds t to the response for the user.

- **Changes to result processing.** After the decryption, the user calculates a set hash over the received values resulting in $resultValuesHash'$, calculates $t' \leftarrow \mathsf{MAC\_Tag}(SK^K, resultValuesHash')$, and accepts the result only if $t' = t$. The MAC creation and verification protect against deviation (3).

Summarizing, all possible deviations are mitigated. Hence, the extension described in this section provides security under an active attacker. Performance measurements show that the described modifications introduce an overhead of about 0.3 ms at a query result set size of 100.

6.5 Implementation

For our experiments, we implemented a HardIDX prototype for both constructions. They are implemented in C/C++ and use the Intel SGX SDK in version 1.1. In the following, we elaborate implementation details, which are important with respect to performance and security.

Side channels. As stated in the attacker model (see Section 6.1.3), we consider three possible side channels: external resource access, page-fault, and cache-timing side channel. Using these side channels, an adversary can observe access patterns to memory with the goal of inferring sensitive information. The first two side channels cannot be mitigated by the implementation as they are inherent to Intel SGX. We further discuss these side channels in our security evaluation (Section 6.6.1).

The cache-timing side channel allows a noisy, but fine-grained observation of memory accesses on cache line level. This observation does not reveal sensitive data directly, but can reveal

sensitive information if an access depends on sensitive data. To protect against cache-timing side-channel attacks, our algorithms and implementation do the following: During the B$^+$-tree creation, every node is filled up with dummy search keys and dummy pointers; all search keys and pointers are padded to a fixed bit-length (32 bit in our implementation); and all nodes and values are stored at a random position. B$^+$-tree searches inside the enclave always access every search key and pointer, even if the search could end once a search key is found that is larger than the search range. By these and other fine-grained implementation details, we achieve data independent accesses and thwart the cache-timing side channel.

Memory Management. In particular construction 2 is optimized with respect to memory transfer operations and context switches between untrusted part and enclave. To reduce the number of context switches, the untrusted part holds a list of all requested nodes. Nodes from this list (up to a specified maximum) are transferred and processed at once, i.e., with only one switch. The memory transfer is optimized by exploiting the fact that the enclave can access the memory of its host process. The B$^+$-tree is loaded in the host process's memory from where the enclave can fetch nodes directly. This is much more efficient than copying nodes explicitly into enclave memory.

Cryptography. As AE implementation, we use AES-128 in GCM mode [49], which is supported by the Intel SGX SDK. It uses leakage resilient implementations and hardware features [197]. For instance, it uses AES-NI hardware which holds the S-Boxes in CPU registers instead of RAM, thus hampering cache side-channel attacks [198], [199]. As set hash function, we use the Set-XOR-Hash construction (see Section 2.2.7).

6.6 Evaluation

Based on the two HardIDX constructions presented in Section 6.3 and the implementation details provided in Section 6.5, this section gives an in-depth security and performance evaluation.

6.6.1 Security Evaluation

First, we evaluate the security of the constructions individually. Then, we compare their leakages. Finally, we discuss the security under an active attacker.

Construction 1. We start by considering two side channels stated in HardIDX's attacker model (see Section 6.1.3): page-fault and cache-timing side channel. We do not discuss the cache-timing side channel as it is mitigated by our implementation (see Section 6.5). Afterwards, we define the leakage under a passive attacker and use it to prove Theorem 1.

For the B$^+$-tree search, the external-resource-access side channel is not relevant, because the whole B$^+$-tree structure is loaded into the enclave memory during the setup phase. Therefore, the search does not require any external accesses. The only external communication is the input of an encrypted query and the output of a shuffled list of result pointers. The query is encrypted with AE and thus an attacker does not learn anything about the query. In particular, AE hides if an equal query was sent before. The leakage of the output pointers is considered in the leakage functions, which we define later in the section.

The page-fault side channel allows an attacker to observe all accesses to individual enclave memory pages, but accesses within the same page are indistinguishable. Each enclave memory page can contain both, code and data, and allocates 4 kB of main memory. Our implementation uses 128 bit AES blocks. Thus, up to $k = 4\,\text{kB}/(o \cdot 128\,\text{bit})$ nodes are contained in one page where o denotes the number of AES blocks used by each node. Our experiments show that $o = 102$ if $T.b = 100$ and 32 bit search keys and pointers are used. Therefore, multiple (huge)

nodes fit within a single page. We use the notation $X \in \rho$ to express that X is stored in page ρ. The B$^+$-tree nodes are stored next to each other in memory, and they fit into $|\boldsymbol{\rho}| = \lceil k/|\boldsymbol{X}| \rceil$ pages $\boldsymbol{\rho} = (\rho_1, \ldots \rho_{|\boldsymbol{\rho}|})$.

Based on this side-channel discussion, the HSBT1 algorithms, the attacker model, and the HSBT CKA2-HW–security definition (see Section 6.3.1), we now define the leakage functions of HSBT1:

$\mathcal{L}^{enc}(\boldsymbol{S})$: Given the key-value pairs $\boldsymbol{S} = ((K_1, V_1), \ldots, (K_n, V_n))$, this function outputs the number of B$^+$-tree nodes $|\boldsymbol{X}|$, the number of indexed values n, and the size of every value.

$\mathcal{L}^{hw}(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$: Given the key-value pairs $\boldsymbol{S}$, the plaintext B$^+$-tree $\boldsymbol{T}$, the search range $\boldsymbol{R}$, and a point in time t, this function outputs the pages access pattern $\Phi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ and the values access pattern $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$.

Loosely speaking, the pages access pattern $\Phi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ is a tree that contains all pages in $\boldsymbol{\rho}$ that get accessed when the range $\boldsymbol{R}$ is searched for. A directed edge in Φ from a parent to a child means that the parent gets accessed before the child. For a more formal definition, we define $\boldsymbol{M}$ as the set of all pages in which leaf nodes are present that contain search keys falling in the search range, i.e., $\boldsymbol{M} = \{\rho \,|\, \rho \in \boldsymbol{\rho} \wedge X \in \rho \wedge X \in \boldsymbol{T} \wedge X.isLeaf \wedge \exists\, j \in [1, T.b-1] : X.K_j \in \boldsymbol{R}\}$. Now, we can specify the node set $\boldsymbol{Y}$ of Φ as $\boldsymbol{Y} = \{\rho_i \,|\, \rho_i \in \boldsymbol{M}\} \bigcup \{\rho_i \,|\, \exists\, l \in [1, T.h] : X_2 == X_1 \rightarrow parent_l \wedge X_1 \in \rho_j \wedge \rho_j \in \boldsymbol{M} \wedge X_2 \in \boldsymbol{T} \wedge X_2 \in \rho_i \wedge \rho_i \in \boldsymbol{\rho}\}$. The edge set of Φ is $\{(\rho_i, \rho_j) \,|\, \rho_i, \rho_j \in \boldsymbol{Y} \wedge \rho_i \neq \rho_j \wedge \exists\, X_1, X_2 \in \boldsymbol{T} : (X_1 \in \rho_i \wedge X_2 \in \rho_j) \wedge X_1 == X_2 \rightarrow parent_1\}$. The time parameter t specifies a random but fixed order of sibling nodes in Φ. See Figure 6.3 for an illustrative example.

The values access pattern $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ is a set of page-pointers tuples. Each tuple is a combination of a page containing leaf nodes containing search keys falling in the search range and the set of the corresponding pointers. More formally, $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t) = \{(\rho, \boldsymbol{P}^\rho) \,|\, \rho \in \boldsymbol{\rho} \wedge X \in \rho \wedge X \in \boldsymbol{T} \wedge X.isLeaf \wedge \exists\, j \in [1, T.b-1] : X.K_j \in \boldsymbol{R} \wedge \boldsymbol{P}^\rho = \{X.P_l \,|\, X \in \rho \wedge l \in [1, T.b-1] \wedge X.K_l \in \boldsymbol{R}\}\}$. The time parameter t specifies a random but fixed order of the pointers.

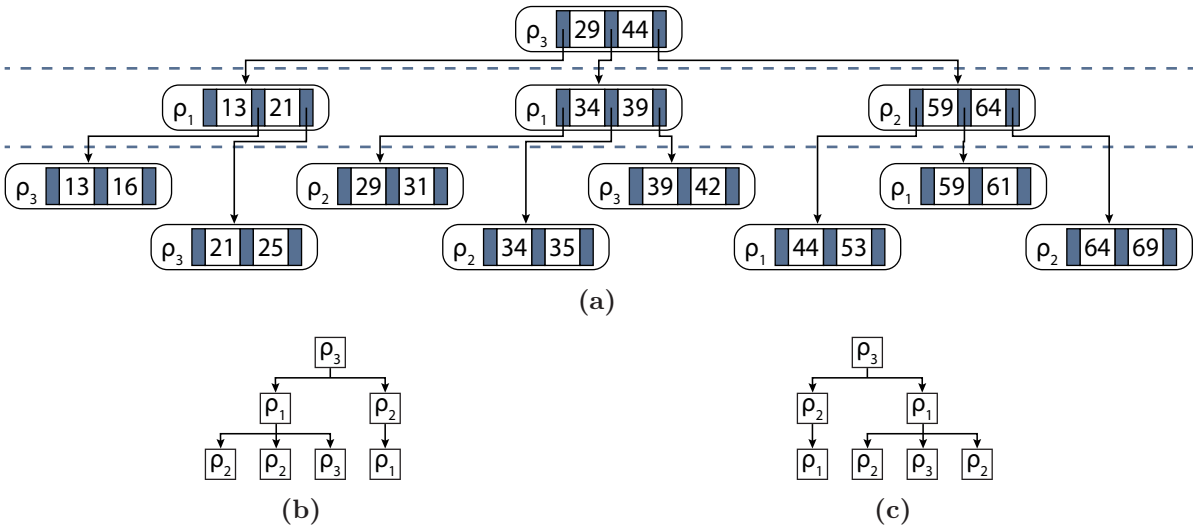

Figure 6.3: Illustration of page access pattern leakage Φ: (a) example B$^+$-tree $\boldsymbol{T}$ with the pages containing the node on the left, (b) leakage $\Phi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ for $\boldsymbol{R} = [33, 55]$ and B$^+$-tree $\boldsymbol{T}$ at t, (c) leakage $\Phi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ for $\boldsymbol{R} = [33, 55]$ and B$^+$-tree $\boldsymbol{T}$ at t'.

The pages access pattern Φ and the values access pattern Ψ are worst-case estimations. An attacker would require many queries to exactly determine which page is a child of another page in Φ. The same applies for the exact matching of result values to a page, which is leaked by Ψ.

Proof of Theorem 1. We describe a PPT simulator $\mathcal{S}$ for which a PPT adversary $\mathcal{A}$ can distinguish $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability:

- *Setup:* $\mathcal{S}$ creates a random key $SK^{rnd} = \mathsf{AE\_Gen}(1^{\lambda})$ and stores it.
- *Simulating γ:* $\mathcal{S}$ gets $\mathcal{L}^{enc}$. It uses the contained number of nodes $|\boldsymbol{X}|$ to create a set of nodes $\boldsymbol{X}^{rnd} = (X_1, \ldots, X_{|\boldsymbol{X}|})$ filled with random search keys and random pointers. $\mathcal{S}$ encrypts each node $X \in \boldsymbol{X}^{rnd}$ with $\mathsf{AE\_Enc}(SK^{rnd}, X)$, assigns each node a unique, random id $X_i.id \in [0, |\boldsymbol{X}| - 1]$, and stores the encrypted nodes at the position $X_i.pos = \mathsf{PRP}(SK^{rnd}, X_i.id)$. As a result, the nodes will be scattered randomly in the pages $(\rho_1, \ldots \rho_{|\boldsymbol{\rho}|})$. Additionally, $\mathcal{S}$ uses the received number of values n and the size of the values to generate n encryptions of random values $\boldsymbol{C} = (C_1, \ldots, C_n)$ using $\mathsf{AE\_Enc}$. $\mathcal{S}$ outputs $\gamma = (\boldsymbol{X}^{rnd}, \boldsymbol{C})$

  All described operations are possible, because the number of nodes $|\boldsymbol{X}|$, the amount n of values, and the size of every value are included in the leakage $\mathcal{L}^{enc}$. In comparison to the tree output by $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$, the simulated γ contains the same number of nodes, the nodes have the same size, and the encrypted values have the same size. The IND-CCA security of AE makes the nodes and values indistinguishable from the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$.
- *Simulating τ:* The simulator $\mathcal{S}$ creates a random range $\boldsymbol{R} = [R^s, R^e]$ and encrypts the range, i.e., $\tau \leftarrow \mathsf{AE\_Enc}(SK^{rnd}, R^s || R^e)$. $\mathcal{S}$ outputs τ.

  The IND-CCA security of AE makes the simulated τ indistinguishable from the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$.
- *Simulating the enclave:* At time t, the simulator $\mathcal{S}$ receives an encrypted range token τ and $\mathcal{L}^{hw}$. $\mathcal{S}$ uses Φ to simulate the page access pattern: it starts at the root of Φ and follows the links unambiguously defined by t. Afterwards, $\mathcal{S}$ creates $\boldsymbol{P}^{res} = \bigcup_{(\rho, \boldsymbol{P}^{\rho}) \in \Psi} \boldsymbol{P}^{\rho}$. Using Ψ, $\mathcal{S}$ determines the order of the pointers in $\boldsymbol{P}^{res}$ and outputs $\boldsymbol{P}^{res}$.

  $\mathcal{A}$ cannot distinguish the page access of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and the simulated access, because the page access pattern delivers deterministic results. Therefore, the results are consistent for different requests of the same range and for queries of distinct or overlapping ranges. Furthermore, the number of result pointers matches and the pointers are consistent, because $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ is unambiguous. The values pointed to are indistinguishable, because they are protected by the IND-CCA security of AE.

□

Construction 2. In this section, we precisely define the leakage of construction 2 under a passive attacker and use it to prove Theorem 2. Regarding the two side channels not handled by our implementation—the page-fault and cache-timing side channels—we only describe the differences compared to construction 1.

The external-resource-access side channel is relevant for the B+-tree search at construction 2, because the enclave accesses individual nodes. Thus, we explicitly consider this leakage in $\mathcal{L}^{hw}$. The page-fault side channel does not leak additional information at construction 2, because nodes are smaller than memory pages. Therefore, this information is already contained in the leakage of the external-resource-access side channel.

Based on this side channel discussion, the HSBT2 algorithms, the attacker model, and the HSBT CKA2-HW–security definition, we now define the leakage functions of HSBT2.

$\mathcal{L}^{enc}(\boldsymbol{S})$: Given the key-value pairs $\boldsymbol{S} = ((K_1, V_1), \ldots, (K_n, V_n))$, this function outputs the number of B+-tree nodes $|\boldsymbol{X}|$, the number of values n, and the size of every value.

$\mathcal{L}^{hw}(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$: Given the key-value pairs $\boldsymbol{S}$, the plaintext B$^+$-tree $\boldsymbol{T}$, the search range $\boldsymbol{R}$, and a point in time t, this function outputs the nodes access pattern $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ and the value access pattern $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$.

The nodes access pattern $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ is a tree that contains the storage positions of all nodes in $\boldsymbol{T}$ that get accessed when searching for the range $\boldsymbol{R}$. A directed edge in $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ from a parent to a child means that the parent gets accessed before the child. For a more formal definition, we denote the set of leaf nodes that contain search keys from the search range by $\boldsymbol{M}$, i.e., $\boldsymbol{M} = \{X \,|\, X \in \boldsymbol{T} \wedge X.isLeaf \wedge j \in [1, T.b-1] : X.K_j \in \boldsymbol{R}\}$. Now, we can specify the node set $\boldsymbol{Y}$ of Υ as $\boldsymbol{Y} = \{X_i.pos \,|\, X_i \in \boldsymbol{M}\} \cup \{X_i.pos \,|\, X_i \in \boldsymbol{T} \wedge X \in \boldsymbol{M} \wedge \exists l \in [1, T.h] : X_i == X{\rightarrow}parent_l\}$. The set of directed edges in Υ is $\{(X_i.pos, X_j.pos) \,|\, X_i.pos, X_j.pos \in \boldsymbol{Y} \wedge X_i == X_j{\rightarrow}parent_1\}$. The time parameter t specifies a random but fixed order of sibling nodes in Υ. See Figure 6.4 for an illustrative example.

The values access pattern $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ is a set of node-pointers tuples. Each tuple is a combination of a leaf node containing search keys falling in the search range and the set of the corresponding pointers. More formally, $\Psi(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t) = \{(X, \boldsymbol{P}^X) \,|\, X \in \boldsymbol{T} \wedge X.isLeaf \wedge \exists\, j \in [1, T.b-1] : X.K_j \in \boldsymbol{R} \wedge \boldsymbol{P}^X = \{X.P_l \,|\, l \in [1, T.b-1] \wedge X.K_l \in \boldsymbol{R}\}\}$. The time parameter t defines a random but fixed order of the pointers.

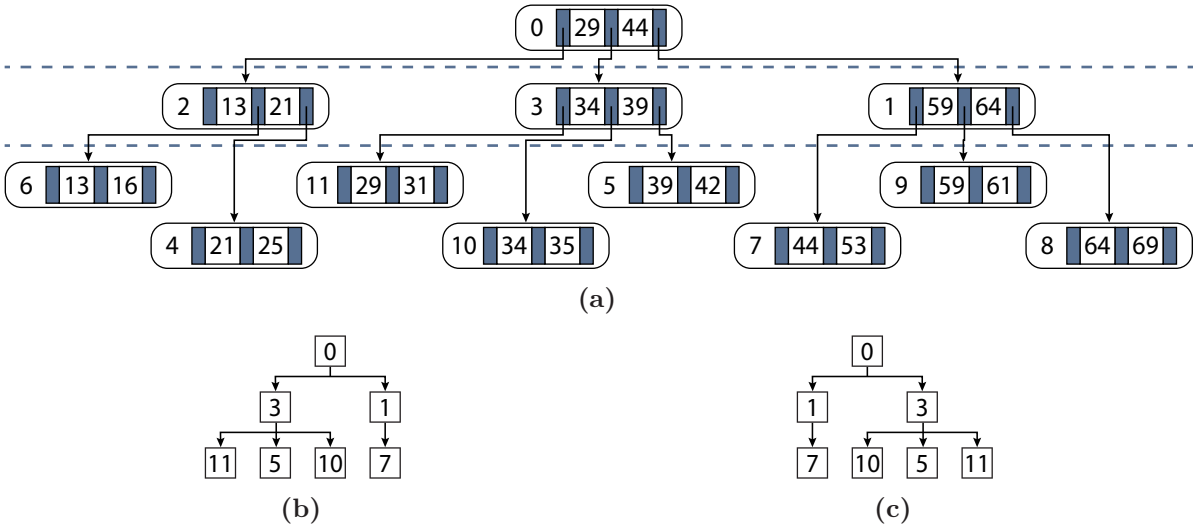

Figure 6.4: Illustration of nodes access pattern leakage Υ: (a) example B$^+$-tree $\boldsymbol{T}$ with the nodes' storage positions on the left, (b) leakage $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ for $\boldsymbol{R} = [33, 55]$ and B$^+$-tree $\boldsymbol{T}$ at t, (c) leakage $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ for $\boldsymbol{R} = [33, 55]$ and B$^+$-tree $\boldsymbol{T}$ at t'.

The nodes access pattern Υ and the values access pattern Ψ are again worst-case estimations.

Proof of Theorem 2. We describe a PPT simulator $\mathcal{S}$ for which a PPT adversary $\mathcal{A}$ can distinguish $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and $\boldsymbol{Ideal}_{\mathcal{A},\mathcal{S}}(\lambda)$ with negligible probability.

- *Setup:* $\mathcal{S}$ creates a random key $SK^{rnd} = \mathsf{AE\_Gen}(1^\lambda)$ and stores it.
- *Simulating γ:* $\mathcal{S}$ gets $\mathcal{L}^{enc}$. It uses the contained number of nodes $|\boldsymbol{X}|$ to create a set of nodes $\boldsymbol{X}^{rnd} = (X_1, \ldots, X_{|\boldsymbol{X}|})$ filled with random search keys and random pointers. $\mathcal{S}$ encrypts each node $X \in \boldsymbol{X}^{rnd}$ with $\mathsf{AE\_Enc}(SK^{rnd}, X)$, assigns each node a unique, random id $X_i.id \in [0, |\boldsymbol{X}| - 1]$, and stores the encrypted nodes at the position $X_i.pos = \mathsf{PRP}(SK^{rnd}, X_i.id)$. As a result, the nodes will be scattered randomly in the memory.

Additionally, $\mathcal{S}$ uses the received number of values n and the size of the values to generate n encryptions of random values $\boldsymbol{C} = (C_1, \ldots, C_n)$ using $\mathsf{AE\_Enc}$. $\mathcal{S}$ outputs $\gamma = (\boldsymbol{X}^{rnd}, \boldsymbol{C})$

All described operations are possible, because the number of nodes $|\boldsymbol{X}|$, the amount n of values, and the size of every value are included in the leakage $\mathcal{L}^{enc}$. In comparison to the tree output by $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$, the simulated γ contains the same number of nodes, the nodes have the same size, and the encrypted values have the same size. The IND-CCA security of AE makes the nodes and values indistinguishable from the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$.

- *Simulating τ:* The simulator $\mathcal{S}$ creates a random range $\boldsymbol{R} = [R^s, R^e]$ and encrypts the range, i.e., $\tau \leftarrow \mathsf{AE\_Enc}(SK^{rnd}, R^s||R^e)$. $\mathcal{S}$ outputs τ.

  The IND-CCA security of AE makes the simulated τ indistinguishable from the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$.
- *Simulating the enclave:* At time t, the simulator $\mathcal{S}$ receives encrypted nodes $\boldsymbol{X}$, an encrypted token τ, and $\mathcal{L}^{hw}$. $\mathcal{S}$ decrypts each node $X \in \boldsymbol{X}$ with $\mathsf{AE\_Dec}(SK^{rnd}, X)$, uses $X.id$ to calculate the storage position, i.e., $X.pos = \mathsf{PRP}(SK^{rnd}, X_i.id)$, and performs a dummy search starting at $X.pos$ with the same access pattern as the actual search. Then, it further processes X based on the leaf attribute:
  - X is not a leaf: $\mathcal{S}$ uses $\Upsilon(\boldsymbol{S}, \boldsymbol{T}, \boldsymbol{R}, t)$ to determine the child nodes that should be requested based on the received node. For each child, it returns a tuple containing $false$ and the pointer to the child in the order defined by t.

    $\mathcal{A}$ cannot distinguish the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ and the simulated output, because the pointers point to indistinguishable nodes according to the IND-CCA security of AE. Furthermore, the results are consistent for different requests of the same range as the nodes access pattern delivers deterministic results and the pseudorandom permutation creates unambiguous positions for the simulated nodes. The same argument applies for queries of distinct or overlapping ranges.
  - X is a leaf: $\mathcal{S}$ uses the leakage Ψ to create a set with all result pointers $\boldsymbol{P}^{res} = \bigcup_{(X, \boldsymbol{P}^X) \in \Psi} \boldsymbol{P}^X$ in the order defined by t. $\mathcal{S}$ combines each value in $\boldsymbol{P}^{res}$ with $true$ and returns the result.

    This output is indistinguishable from the output of $\boldsymbol{Real}_{\mathcal{A}}(\lambda)$ as the number of result pointers matches and the pointers are consistent because Ψ is unambiguous. The values pointed on are indistinguishable, because they are protected by the IND-CCA security of AE.

□

Leakage comparison. The main difference in the leakages of construction 1 and construction 2 is the granularity of the leakages. In construction 1, the attacker can reveal accesses on a page level, because Intel SGX inherently leaks the page access pattern. In construction 2, the attacker can reveal accesses on a node level, because external accesses are necessary. This means that construction 2 leaks data on a finer granularity.

Active attacker. We omit the correctness and CKA2-HW–security definitions under an active attacker for brevity, but they are easily deducible from Definition 31 and Definition 32. We further only consider construction 2 under an active attacker, but the arguments and techniques can be applied to construction 1 with minor modifications.

We identified two basic attack vectors that cover a wide range of possibilities: First, an active attacker can try to attack the protection mechanisms of Intel SGX to gain insights about data and algorithm execution not under her control. According to our attacker model (see Section 6.1.3), we have to protect against three side-channel attacks. As explained before, we mitigate one with our implementation and the other two are covered by the leakage functions

of constructions 1 and 2. For other attacks, we rely on Intel SGX's protection mechanism to guarantee security and correctness under an active attacker.

Second, an active attacker can try to influence the data and protocol execution under her control to gain additional sensitive information or to prevent the user from getting the correct result:

- **Unprotected static data:** The only static data influenceable by the active attacker are values and nodes. The usage of AE guarantees the security of this data. The AE scheme also thwarts correctness attacks trying to modify static data, because these actions are noticed by the decryption algorithm. However, an attacker might delete nodes or values.
- **Unprotected dynamic data:** The only dynamic data influenceable by the attacker are search tokens. The security and integrity of search tokens are guaranteed by AE. A replay attack does not provide the attacker with any additional information as the tree is static and the leakage stays the same for a replayed token. The attacker could drop tokens, but such a DoS attack is out of scope according to our attacker model.
- **Unprotected algorithms:** HSBT2_SearchRange is the only algorithm under attacker control. Besides DoS, there is only a fixed set of possible protocol deviations. Modified static or dynamic data is directly noticeable by the enclave, which can reject further processing. Reducing the number of nodes passed to the enclave at once slows down the process, but does not lead to additional information and does not impact the correctness. In fact, passing one node after another is already covered by the defined leakage. Passing nodes to the enclave that are not expected does not leak additional information, as this information is leaked by the enclave not requesting them. The only remaining deviations are the following: (1) do not pass the root node first to the enclave, (2) do not pass all requested nodes to the enclave, and (3) do not pass all results to the user. Note that the open issue discussed for unprotected static data—node and value deletions—have the same result as deviation (2) and (3).

Overall, the deviations (1)–(3) are the only possibilities of an active attacker that we consider. A mechanism to protect against these deviations is presented in Section 6.4.2. It is important to note that the deviations do influence the correctness of the protocol, but not the security. Thus, the security proofs provided for passive attackers are also valid for active attackers.

6.6.2 Performance Evaluation

In this section, we present performance results of multiple experiments. First, we compare our two constructions described in the design section. Then, we examine the effect of the constructions' different memory-management strategies. Finally, we compare our solution against the currently[3] fastest, polylogarithmic range query search algorithms [149], [183].

We performed all experiments with one Intel SGX–enabled Intel Core i7-6700 @ 3.40 GHz and 32 GB DDR4 RAM. We further used 64-bit Ubuntu 14.04.1 extended with Intel SGX support. All presented latencies measure the processing time spent at the server excluding any network delay.

Construction 1 vs. construction 2. For this experiment, we set the branching factor $T.b$ to 10 and the tree contains 1,000,000 key-value pairs. We perform 1000 random range queries $\boldsymbol{R} = [R^s, R^e]$ with five result set sizes: $2^0, 2^4, 2^8, 2^{12}, 2^{16}$. In more detail, a range start R^s is selected uniformly at random from the value domain. Next, R^e is set to the search key that comes $2^0, 2^4, \ldots$ entries later in a sorted list of the search keys contained in the tree. If R^s is

3 At the time of the publication of the paper [42] corresponding to this chapter.

not followed by enough search keys, the sampling is repeated. The following range searches have a result set size of 2^0 in the example depicted in Figure 6.1: $[2, 4]$, $[40, 40]$, and $[60, 82]$.

Figure 6.5 depicts the results of the described experiment, whereby the x-axis shows the size of the result set and the y-axis shows the mean query latencies. Construction 2 is slower than construction 1 by a small factor at any result set sizes. The performance difference can be explained by the following effects:

- **Context switch.** On each call into and out of the enclave, Intel SGX performs a context switch. As we explain in detail in Section 3.1.2, this switch includes, e.g., storing processor state, restoring processor state, and copying data. Construction 1 only requires one context switch. In contrast, construction 2 requires $O(T.h)$ switches, as at least each level of the B$^+$-tree is loaded into the enclave individually.
- **Data transfer.** In construction 1, the data transfer between untrusted part and enclave is limited to the incoming query and the outgoing result set. In contrast, construction 2 additionally transfers individual nodes between untrusted part and enclave.
- **Access to plaintext.** In construction 1, decryption is a one-time effort done in the setup phase. During query processing, construction 1 has plaintext access to all nodes of the B$^+$-tree. Construction 2 incrementally loads the B$^+$-tree nodes from untrusted storage and decrypts all processed nodes individually.

For an increasing size of the result set, both algorithms search a linearly increasing part of the tree. The latencies of our two constructions converge (on a logarithmic scale). This shows that the effects described above diminish compared to the search time of the algorithm.

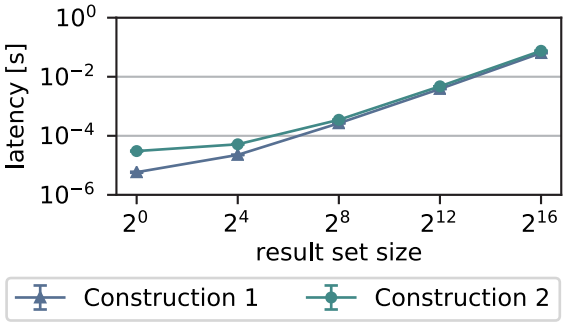

Figure 6.5: Comparison of constructions (95% confidence interval).

Memory management. In order to identify the limiting parameters in the memory management of our two constructions, we evaluate B$^+$-trees with different tree sizes, i.e., number of key-value pairs, and branching factors. For both constructions and for each tree size, we run 1000 randomly chosen queries with a fixed result set size of 100 and test the branching factors 10, 25, 50 and 100. The results for the two constructions are depicted in Figure 6.6a and Figure 6.6b. The x-axis shows the size of the B$^+$-tree and the y-axis shows the mean query latencies.

In Figure 6.6a, we see a sharp increase of the latency above a tree size of 10^6. This is due to the exhausted EPC memory and the virtual memory mechanism of the OS that swaps pages in and out. This is not security critical since pages remain encrypted and integrity protected by Intel SGX, even when they are swapped out of the EPC.

We see a significant difference in the impact of paging between the different branching factors. This becomes clear by considering the number of required page swaps. The lower the branching factor of a B$^+$-tree, the higher the number of nodes (scattered in main memory). The higher the number of nodes, the higher the number of accesses to different memory pages. The higher

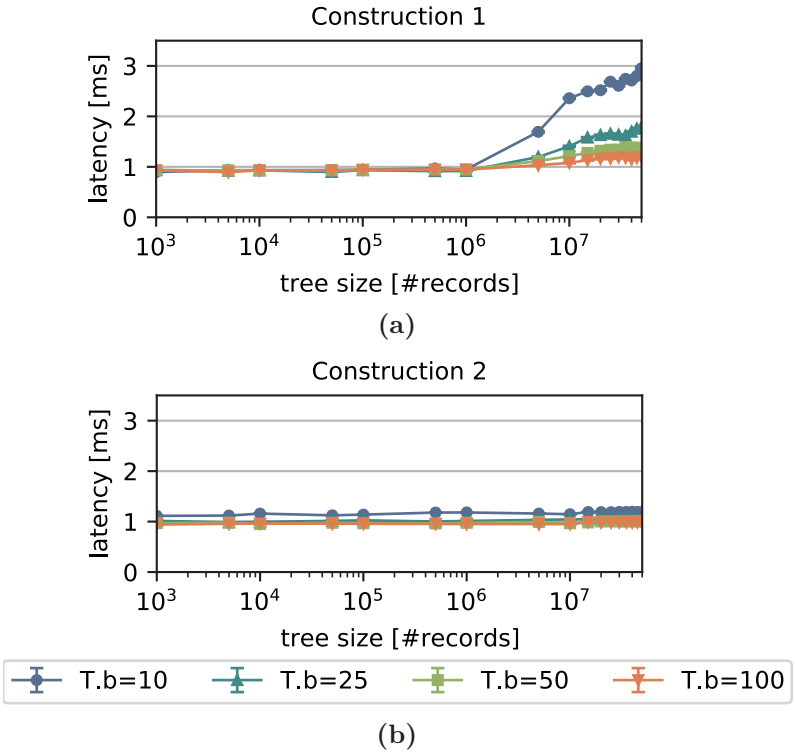

Figure 6.6: Effect of different branching factors in (a) construction 1 and (b) construction 2 with 95% confidence interval.

the number of different page accesses, the higher the probability of an access to a swapped-out page.

In Figure 6.6b, we see that construction 2 is not affected by paging albeit supporting an unlimited tree size. Our data also shows that, as expected, higher branching factors result in better performance. Disregarding the paging problem of construction 1 above a tree size of 10^6 records, a direct comparison of the constructions reveals that the latency of construction 2 approaches the latency of construction 1 for higher branching factors.

Comparison with related work. As final experiment, we compare construction 2 against the currently fastest approach with comparable security features and a security proof presented by Demertzis *et al.* [149]. The authors present seven constructions that support range queries. The constructions have different trade-offs regarding security, query size, search time, storage, and false positives. We do not compare against the highly secure scheme with prohibitive storage cost and also exclude the approaches with false positives as construction 2 does not lead to false positives. Instead, we compare against the most secure approach without these problems: Logarithmic-URC. Demertzis *et al.* do not determine a SSE scheme on which Logarithmic-URC is based on. However, they use the OXT construction from Cash *et al.* [194] for security and performance evaluation. Thus, our implementation also uses OXT as SSE scheme.

Faber *et al.* [183] present a construction quite equal to Logarithmic-URC. We implemented the algorithm of Demertzis *et al.*, but a security and performance comparison to the construction of Faber *et al.* would lead to comparable results.

For our comparison between Logarithmic-URC and construction 2, we search for random ranges with a result set size of 100 and repeat each test 1000 times with four tree sizes: 100, 1000, 10,000, and 100,000. At construction 2, we use a branching factor $T.b$ of 100. Table 6.2 shows the results of this experiment.

Tree Size	100	1000	10,000	100,000
Logarithmic-URC	0.015 s	0.020 s	0.051 s	1.052 s
Construction 2 ($T.b = 100$)	0.119 ms	0.121 ms	0.124 ms	0.125 ms

Table 6.2: Latency comparison of random range queries with Logarithmic-URC [149] and construction 2.

Construction 2 runs in about a tenth of a millisecond and with a very moderate increase for all tree sizes. In contrast, Logarithmic-URC requires at least multiple milliseconds up to a second for bigger trees. A reason for the performance difference might be that OXT construction itself is less efficient than our construction. Furthermore, the search time of OXT depends on the number of entries. Logarithmic-URC fills the OXT construction with elements from a binary tree over the search key domain $\mathcal{D}$ for every stored search key. An increasing domain severely increases the tree height of a binary tree and thus the number of entries for OXT. In contrast, the height of the B^+-tree in our construction is independent of the domain size. The height only grows in the number of search keys and is dependent on the branching factor.

Besides the latency difference, our construction only requires index storage in $O(n)$ and Logarithmic-URC requires $O(n \log \mathcal{D})$. A functional difference between Logarithmic-URC and construction 2 is that Logarithmic-URC requires to fix the search key domain $\mathcal{D}$ beforehand. In our constructions, it is not necessary to fix the domain and the domain size has no performance implications.

It is not trivial to compare Logarithmic-URC and construction 2 regarding security. The access pattern leakage and the leakage of the internal data structure of Logarithmic-URC are comparable to our access pattern leakages. However, Logarithmic-URC additionally leaks the domain size, the search range size, and the search pattern. The search pattern reveals whether the same search was performed before, which might be sensitive information.

6.7 Summary

In this chapter, we introduced how TEE-protected B^+-trees can be used for a secure, efficient, outsourced data processing system. This system—HardIDX—supports range searches and equality searches over encrypted data using a TEE. HardIDX is also deployable as a secure index in an encrypted database. We presented two HardIDX constructions with different memory-management strategies. For both constructions, we presented leakage functions under a passive attacker, explicitly including side channels aspect of Intel SGX. We provided formal security proofs for the claimed leakages and showed how to secure HardIDX in an active attacker environment. HardIDX's TCB is small exposing a small attack surface. The performance evaluation showed that range queries over 50 million encrypted B^+-tree entries require about 1 ms if the result size is 100. This demonstrates that HardIDX scales to large tree sizes. In contrast, the fastest related work requires more than a second for 100,000 encrypted entries.

7 Protected Database Dictionaries: EncDBDB

In this chapter, we examine the secure, outsourced, TEE-based processing of the *dictionary* data structure. To be precise, we examine dictionaries that are used for dictionary encoding of database columns. Dictionary encoding splits each database column into two structures: a dictionary containing a list of (potentially unique) values and an *attribute vector* containing references to dictionary entries according to the original column. The main goal of dictionary encoding is data compression. The compression rate is high if the original column has many values in total, but only a few unique values.

Using TEE-protected dictionaries, we design EncDBDB—a secure, efficient, outsourced, dictionary-encoding–based, column-oriented, in-memory database supporting analytic queries on large datasets. Dictionary encoding reduces the storage space overhead of large (encrypted) datasets [200], [201], column-oriented data storage optimizes the processing of analytic workloads [202]–[205], and in-memory processing boosts the overall performance [206]–[208].

Therefore, EncDBDB is especially suited for data warehouses, which are used by companies for business intelligence and decision support. Such warehouses contain large datasets and the underlying DBMSes are optimized for complex, read-oriented, analytic queries. In this chapter, we focus on one complex, required query type: range queries. Equality queries are supported implicitly, as they can be expressed as range searches. Additionally, we explain how EncDBDB can handle joins, insertions, deletions, updates, counts, aggregations, and average calculations.

EncDBDB offers nine encrypted dictionary types, which provide different security, performance, and storage efficiency trade-offs for the stored data. The data owners can select an arbitrary encrypted dictionary type for each database column according to their desired requirements.

Our evaluation describes the security, performance, and storage efficiency trade-off of the different encrypted dictionary types. In the security evaluation, we first compare the security of six encrypted dictionary types with security schemes known in literature. Then, we classify the security of the remaining three encrypted dictionary types relative to the six others. Using an Intel SGX-based EncDBDB prototype and a real-world dataset, we perform multiple storage and performance experiments. Finally, we provide a usage guideline for the different encrypted dictionary types.

An encrypted database can also be built using purely cryptographic protection. For instance, FHE [17] supports arbitrary computations on encrypted data. However, FHE is too slow for an efficient encrypted database [19], [180]. CryptDB [182] and Monomi [186] use multiple encryption schemes, e.g., DET [10], [125], OPE [11], [12], [127], [129], and ORE [127], [132], [133], to perform different database functionalities. The encryption schemes are layered and/or stored in parallel, introducing a storage overhead. Additionally, careful query rewriting is necessary to receive a result securely and efficiently. Alternatively, a TEE can be used to build an encrypted database, but related TEE-based approaches assume an enclave memory size that is not supported efficiently by available TEEs [164], [209], do not provide DBMS functionality [42], do not support persistency [30], or leak the result of every primitive operation [29]. Also, all mentioned TEE-based approaches do not consider data compression to reduce the size of large databases.

The contributions of this chapter are the following:

- A secure, efficient, outsourced, TEE-based, column-oriented, in-memory database using protected dictionaries.
- Nine encrypted dictionary types from which data owners can freely select per column according to their requirements. The nine types provide different security (order and frequency leakage), performance, and storage efficiency trade-offs. The security ranges from the equivalent of deterministic ORE [132] to *range predicate encryption* (RPE) [148].
- Integration of the EncDBDB approach into MonetDB [210]–[212], an open source DBMS. The enclave has only 1129 LOC, reducing the potential for security-relevant implementation errors and side-channel leakages. Query optimization and auxiliary database functionalities, e.g., storage, transaction, and database recovery management still operate without changes to the original code.
- Sub-millisecond overhead for encrypted range queries compared to plaintext range queries, on a real-world customer database containing millions of entries.
- Less storage space required for a compressed, encrypted column with the appropriate encrypted dictionary type than for a plaintext column with the same data.

The remainder of this chapter is structured as follows: In Section 7.1, we give an in-depth introduction into dictionaries; discuss the benefits of dictionary-encoding–based, column-orientated, in-memory databases; explore EncDBDB in detail assuming the existence of multiple encrypted dictionary types; and present the attacker model. In Section 7.2, we discuss approaches related to EncDBDB. Afterwards, we present the design of the nine encrypted dictionary types in Section 7.3. In Section 7.4, we present an EncDBDB extension enabling EncDBDB to handle dynamic data, followed by relevant implementation details of our EncDBDB prototype in Section 7.5. In Section 7.6, we provide an in-depth security, storage, and performance evaluation for the different encrypted dictionary types. In the same section, we additionally present a usage guideline for the encrypted dictionary types. We conclude this chapter with a summary in Section 7.7.

7.1 Design Considerations

Our goal is to use TEE-protected dictionaries to build a secure, efficient, outsourced, dictionary-encoding–based, column-oriented, in-memory database. In this section, we first give an in-depth introduction of the data structure considered in this chapter: dictionaries. Next, we discuss the benefits of combining in-memory processing, column-orientated storage, and dictionaries. Then, we give an overview of the considered system called EncDBDB. Finally, we present the assumed attacker model.

7.1.1 Data Structure: Dictionary

A mechanism denoted dictionary encoding can be used by databases for data compression. The idea of dictionary encoding is to split a column $\boldsymbol{C} = (C_0, \ldots, C_{|\boldsymbol{C}|-1})$ into two structures: a dictionary $\boldsymbol{D}$ and an attribute vector $\boldsymbol{AV}$. The dictionary $\boldsymbol{D} = (D_0, \ldots, D_{|\boldsymbol{D}|-1})$ is filled with all values $V \in \boldsymbol{C}$ and every V has to be present in $\boldsymbol{D}$ at least once. The index i of a dictionary entry D_i is called its *ValueID* (*vid*). The attribute vector $\boldsymbol{AV} = (AV_0, \ldots, AV_{|\boldsymbol{AV}|-1})$ is constructed by replacing all values $V \in \boldsymbol{C}$ with one *vid* that corresponds to V. As a result, $\boldsymbol{AV}$ contains $|\boldsymbol{AV}| = |\boldsymbol{C}|$ ValueIDs. The index j of an attribute vector entry AV_j is called its *RecordID* (*rid*). $\mathsf{un}(\boldsymbol{C})$ denotes the set of unique values in $\boldsymbol{C}$, $|\mathsf{un}(\boldsymbol{C})|$ the number of unique values in $\boldsymbol{C}$, $\mathsf{oc}(V, \boldsymbol{C})$ the occurrence indices of a unique value V in $\boldsymbol{C}$, and $|\mathsf{oc}(V, \boldsymbol{C})|$ the number of occurrences of V in $\boldsymbol{C}$. We define the correctness of a column split as follows:

Definition 33 (Split Correctness)**.** *A split of column* $\boldsymbol{C}$ *into a dictionary* $\boldsymbol{D}$ *and an attribute vector* $\boldsymbol{AV}$ *is correct if* i *is the ValueID stored in the attribute vector at position* j *and* D_i *equals* C_j*, i.e.,* $\forall\, j \in [0, |\boldsymbol{AV}| - 1]\colon (i = AV_j \iff D_i = C_j)$.

In Figure 7.1, we present a dictionary encoding example based on a first name column (`FName`). For instance, Jessica is inserted in the dictionary at the ValueID 1 and all positions from the original column that contain Jessica are replaced by this ValueID in the attribute vector (see RecordIDs 0, 2, and 3). The set of unique values is $\mathsf{un}(\boldsymbol{C}) = \{\text{Hans}, \text{Jessica}, \text{Archie}\}$, the number of unique values is $|\mathsf{un}(\boldsymbol{C})| = 3$, Archie occurs at the indices $\mathsf{oc}(\text{Archie}, \boldsymbol{C}) = \{1, 5\}$, and Archie occurs $|\mathsf{oc}(\text{Archie}, \boldsymbol{C})| = 2$ times.

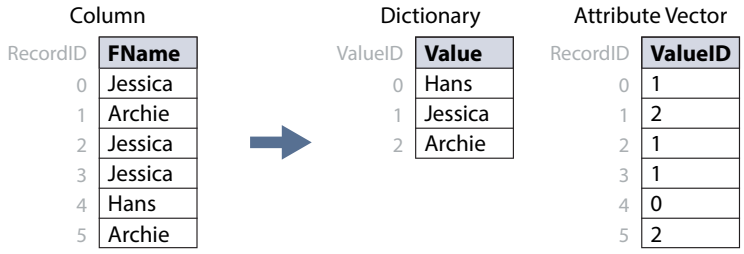

Figure 7.1: Dictionary encoding example.

Dictionary encoding reduces the storage space of a column in many cases, because a ValueID of i Bits is sufficient to represent 2^i different values in the attribute vector and the (variable-length) values only have to be stored once in the dictionary. For instance, a column that contains 10,000 strings of 10 characters each, but only 256 unique values, requires $256 \cdot 10\,\text{B}$ for the dictionary and $10{,}000 \cdot 1\,\text{B}$ for the attribute vector. In total, dictionary encoding reduces the required storage from 100,000 B to 12,650 B in this case. Dictionary encoding achieves the best compression rate if columns contain few unique but many frequent values, because every value has to be stored only once. The real-world data used for our performance evaluation (see Section 7.6.3) and other studies [213], [214] show that this is a characteristic of many columns in data warehouses. The high compression rates achieved by dictionary encoding sparingly use the scarce resource of in-memory databases—main memory.

When using dictionary encoding, two steps are necessary for a range search: a *dictionary search* followed by an *attribute vector search*. The dictionary search checks for each $V \in \boldsymbol{D}$ if it falls into range $\boldsymbol{R}$ and returns the matching ValueIDs $\boldsymbol{vid}$. The attribute vector search linearly scans the attribute vector searching for each value $V \in \boldsymbol{vid}$ and returns a list of matching RecordIDs $\boldsymbol{rid}$. This scan is parallelizable with a speedup expected to be linear in the number of threads.

In the example of Figure 7.1, a dictionary search for $\boldsymbol{R} = [\text{Archie}, \text{Hans}]$ returns $\boldsymbol{vid} = \{0, 2\}$. The corresponding attribute vector search returns $\boldsymbol{rid} = \{1, 4, 5\}$

7.1.2 Dictionary-encoding–based, Column-oriented, In-memory Databases

In-memory database. Many commercial and open source DBMS vendors offer in-memory databases for analytical data processing, e.g., SAP HANA [215], Oracle RDBMS [216], and MonetDB [217]. In-memory databases permanently store the primary data in main memory and use the disk as secondary storage. The major benefit of in-memory databases is the lower access time of main memory compared to disk storage. This speeds up every data access for which disk access would be necessary. Additionally, the fast memory accesses lead to shorter locking times in concurrency control; thus, fewer cache flushes and a better CPU utilization. We refer to the literature for more details about in-memory databases [206]–[208].

Column-oriented, In-memory Database. One possible database storage concept is column-oriented storage, i.e., successive values of each column are stored consecutively (in main memory or on disk), and surrogate identifiers are (implicitly) introduced to connect the rows [202]–[205]. The combination of in-memory databases and column-oriented storage reduces the number of cache misses, which strongly influences the in-memory performance. All in-memory databases mentioned above support column-oriented storage.

The main drawbacks of column-oriented storage are the following: (1) so-called tuple reconstruction is necessary to reassemble a projection involving multiple attributes and (2) any modification of a whole tuple accesses non-contiguous storage locations. These problems are not severe at analytical applications, e.g., data warehousing and business intelligence, because analytical queries often involve a scan on a significant number of all tuples, but only a small subset of all columns [203], [218]. Additionally, bulk loading of data is often used in this context and complex, long, read-only queries are executed afterwards [205], [219]. An example query is a report on total sales per country for products in a certain price range. Only the few columns that are involved in the query have to be loaded and they can be processed sequentially, which is beneficial as it decreases CPUs' cache misses.

Dictionary-encoding–based, Column-oriented, In-memory Databases The three commercial DBMSes mentioned above and many other databases use data compression mechanisms to exploit redundancy within data [200], [201]. Abadi *et al.* [200] study multiple database compression schemes, e.g., null suppression, run-length encoding, and dictionary encoding, and show how these schemes can be applied to column-oriented databases. According to the authors, column-oriented databases in particular profit from compression. In this chapter, we only consider dictionary encoding, because it is the most prevalent compression used in column-oriented databases [200]. High compression rates achieved by dictionary encoding sparingly use the scarce resource of in-memory databases—main memory.

7.1.3 System: EncDBDB

EncDBDB is an encrypted, column-oriented, dictionary-encoding–based, in-memory database. It offers nine encrypted dictionary types, which provide different security, performance, and storage efficiency trade-offs. Additionally, EncDBDB supports nine plaintext dictionary types. These use the same algorithms as the encrypted dictionary types, but the algorithms operate on plaintext data and are not executed in an enclave. We only consider the plaintext dictionary types in our performance evaluation as the focus of this dissertation is on encrypted data processing. Three operations differ for the encrypted dictionary types:

1. Encrypted dictionary creation.
2. Dictionary search.
3. Attribute vector search.

Details about those operations are presented in the design section (Section 7.1). In this section, we consider the different encrypted dictionary types as an existing building block and present how they are used by EncDBDB.

Figure 7.2 presents an overview of EncDBDB's architecture and the process flow, which involve four entities: a trusted data owner, a trusted proxy, a trusted application, and an untrusted cloud server with a trusted (Intel SGX) enclave. In the following, we give an overview of EncDBDB's setup and query phase. For this overview, we only discuss range queries on a static database, i.e., the data is outsourced at one point in time. As (in)equality selects and greater/less than selects can be expressed as range queries, they are covered implicitly. In Section 7.4, we present how EncDBDB handles joins, insertions, deletions, updates, counts, aggregations, and average calculations.

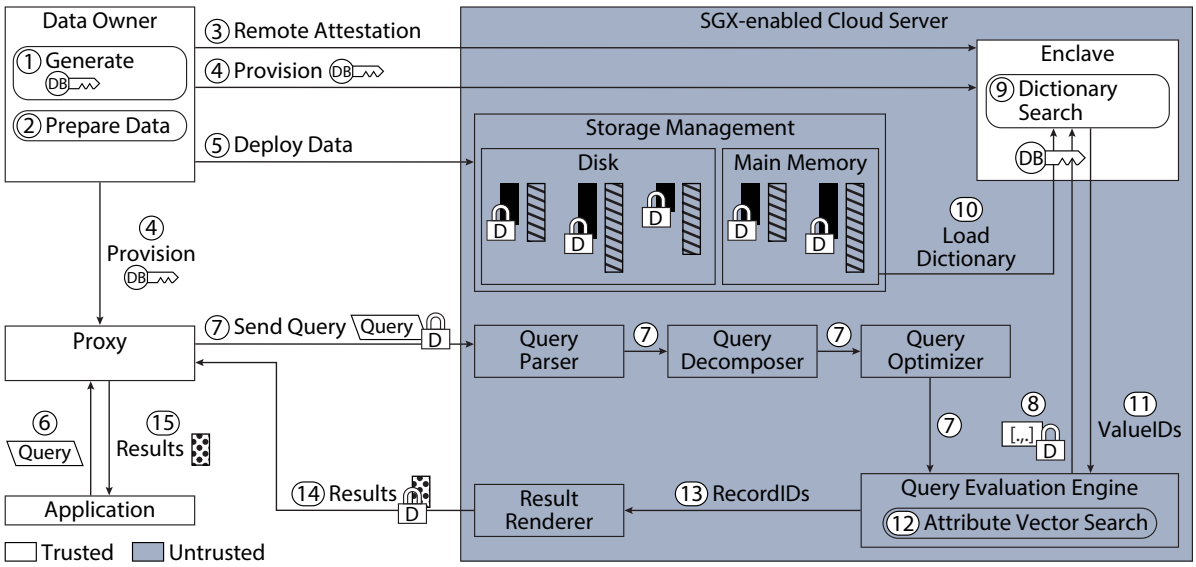

Figure 7.2: Overview and process flow of EncDBDB.

Setup phase. In one possible EncDBDB variant, the cloud provider is assumed trusted for the initial setup. The data owner can upload plaintext columns and the cloud provider can support the data owner in choosing an appropriate encrypted dictionary type for each column. For instance, the cloud provider can guide the data owner with questions about the data sensitivity. Afterwards, the cloud provider performs the corresponding column splits and encryptions. We, however, discuss another variant in which the cloud provider is untrusted also during the setup phase and plaintext data never leaves the realm of the data owner. The setup phase is only executed once and consists of the following steps:

① The data owner defines a security parameter λ and generates a secret key $SK^{DB} = \mathsf{AE}\_\mathsf{Gen}(1^{\lambda})$.

② The data owner takes its plaintext database $\boldsymbol{PDB}$ and selects one encrypted dictionary type for each column $\boldsymbol{C} \in \boldsymbol{PDB}$. Based on the selected encrypted dictionary type, the data owner performs the column split. Then, the data owner encrypts the resulting dictionary with an individual key SK^{D}, which is derived from SK^{DB}, the table name, and the column name, using a key derivation function. The result is an encrypted database $\boldsymbol{EDB}$.

③ The data owner uses Intel SGX's remote attestation feature (see Section 3.1.3) to authenticate the cloud server's enclave and to establish an authenticated channel to the enclave.

④ The data owner uses the authenticated channel to provision SK^{DB} to the enclave. Additionally, the data owner provisions SK^{DB} to the proxy via a secure out-of-band mechanism.

⑤ As a last step of the setup, the data owner uses an import functionality of the cloud provider to deploy $\boldsymbol{EDB}$. The storage management of the in-memory database stores all data on disk for persistency and loads (parts of) it into main memory.

Query phase. From this point on, the application can send an arbitrary number of queries, which are processed as follows:

⑥ The application issues an SQL query Q to the proxy. W.l.o.g. we assume that Q selects and filters only one column. The filter can be an equality select, an inequality select, a

greater than select (inclusive or exclusive), a less than select (inclusive or exclusive) or a range select (inclusive or exclusive). The proxy converts all filters to a range select with range $\boldsymbol{R}$. For instance, the SQL query `SELECT FName FROM t1 WHERE FName <= 'Ella'` is converted to `SELECT FName FROM t1 WHERE FName >=` $-\infty$ `and FName <= 'Ella'` where $-\infty$ is a placeholder for the smallest domain value. Next, the proxy derives SK^D using SK^{DB}, the table name, and the column name. Afterwards, the proxy encrypts the range start and end (R^s and R^e) with AE_Enc under SK^D. The resulting encrypted query eQ of our SQL example is `SELECT FName FROM t1 WHERE FName >=` AE_Enc(SK^D, $-\infty$) `and FName <=` AE_Enc(SK^D, `'Ella'`). Because of the query conversion, the untrusted cloud provider cannot differentiate query types, and because AE is a probabilistic encryption, the cloud provider also cannot learn if the values were queried before.

⑦ The proxy sends the encrypted query eQ to the cloud provider, where eQ is handled by a DBMS-specific query pipeline. For instance, the query is processed by a query parser, a query decomposer, and a query optimizer. The query optimizer selects a query plan and shares it with a query evaluation engine. As only one column is filtered in our example, the plan contains one encrypted dictionary $\boldsymbol{eD}$, one plaintext attribute vector $\boldsymbol{AV}$, and one encrypted range filter τ that has to be executed.

⑧ The query evaluation engine enriches $\boldsymbol{eD}$ with metadata: the table name, the column name, and the column size. Then, it passes τ and a reference to $\boldsymbol{eD}$ to the enclave.

⑨ Depending on the encrypted dictionary type of the filtered column, the enclave performs a specific dictionary search.

⑩ During this search, the enclave loads the necessary dictionary entries from the untrusted realm.

⑪ Finally, the enclave returns a list of ValueIDs $\boldsymbol{vid}$ for which the corresponding values fall into the search range $\boldsymbol{R}$.

⑫ The query evaluation engine performs an attribute vector search corresponding to the encrypted dictionary type of the filtered column.

⑬ The query evaluation engine passes a list of RecordIDs $\boldsymbol{rid}$ to the result renderer.

⑭ The result renderer creates one encrypted result column $\boldsymbol{eC}$ by undoing the column split, i.e., $\boldsymbol{eC} = (eD_j \mid j = AV_i \wedge i \in \boldsymbol{rid})$. Additionally, it enriches $\boldsymbol{eC}$ with column metadata—the table and column name. If a filter query were executed on other columns in the same table, the result renderer would use $\boldsymbol{rid}$ to prefilter these columns. It would also use $\boldsymbol{rid}$ for projections. Finally, the result renderer passes $\boldsymbol{eC}$ back to the proxy.

⑮ The proxy receives one encrypted column $\boldsymbol{eC}$ from the cloud provider and uses the attached column metadata to derive the column specific key SK^D. Every entry in $\boldsymbol{eC}$ is decrypted individually with SK^D resulting in one plaintext column $\boldsymbol{C}$, which is passed back to the application.

Notably, only a very small part of the query processing is done inside the trusted enclave, and the required enclave memory is limited. In particular, it is independent of the dictionary and attribute vector size. There is no need to modify auxiliary database functionalities such as persistency management, multiversion concurrency control, or access management. Still, the complete processing is protected.

7.1.4 Attacker Model

We consider the data owner, the proxy, and the application, which uses the database, as trusted. An untrusted cloud provider deploys EncDBDB on a TEE-enabled machine. The TEE supports the capabilities listed in Table 3.1. We assume an honest-but-curious attacker, i.e., a passive

attacker who follows the protocol, but tries to gain as much information as possible. Except the enclave, the attacker can observe all software running at the cloud provider, e.g., the OS, the firmware, and the DBMS. As a result, the attacker has full access to data stored on disk and in main memory, and she can observe the access pattern to them. Additionally, she can track all communication between the enclave and resources outside of it, and all network communication between the proxy and the DBMS. This includes the incoming queries in which only the data values are encrypted. The enclave is assumed not to have intentional data leakage.

As we show in Section 3.2, Intel SGX is vulnerable to various attacks and multiple mitigations are known. We consider attacks on Intel SGX and their mitigations an orthogonal problem and do not consider them in this chapter. Nevertheless, EncDBDB has a minimal enclave size; therefore, the presented mitigations should be straightforward to integrate. Hardware and DoS attacks are out of scope.

We assume that the attacker targets each database column independently, i.e., she does not use correlation information to target columns. It remains future work to evaluate how decorrelation of columns protects the database in practice.

7.2 Related Work

In this section, we compare EncDBDB to TEE-based, encrypted databases; software-only, encrypted databases; and searchable encryption.

7.2.1 TEE-based, Encrypted Databases

In the following, we outline TEE-based approaches ranging from large to small enclave sizes, and classify EncDBDB accordingly. In Table 7.1, we provide a comparison of existing TEE-based encrypted database approaches and EncDBDB.

Approach	Optimized for Workload	Protection Object
EnclaveDB [209]	OLTP	In-memory storage and query engine
ObliDB [30]	OLTP & OLAP	Data structure (array or B^+-tree)
StealthDB [29]	OLTP	Primitive operators (e.g., $\leq$, $\geq$, $+$, $*$)
EncDBDB	OLAP	Data structure (dictionary)

Approach	Compression	Overhead		TCB LOC
		Storage	Performance	
EnclaveDB [209]	○	N/A	$> 20\,\%$	∼235,000
ObliDB [30]	○	$> 100\,\%$	$> 200\,\%$	∼10,000
StealthDB [29]	○	$> 300\,\%$	$> 20\,\%$	∼1500
EncDBDB	●	$< 100\,\%$	$\sim 8.9\,\%$	1129

Table 7.1: Comparison of related TEE-based, encrypted databases and EncDBDB. The overheads compare the respective approach with a plaintext database. We present lower bounds of the overheads to the advantage of the approaches, taken from the corresponding papers where available. The symbols represent that a feature is supported (●) or not supported (○). The last column presents the lines of code of the trusted computing base.

Haven [164] and SCONE [67] protect entire applications in an untrusted environment using Intel SGX (see Section 4.8.1). Such an application can also be an off-the-shelf DBMS. However,

a complete DBMS with millions of LOC is prone to security-relevant implementation errors or side-channel leakages that could leak arbitrary data from the enclave. Furthermore, no TEE on the market does support the huge enclaves that are necessary for this concept.

Priebe *et al.* propose EnclaveDB [209], a protected database engine that uses Intel SGX to provide confidentiality, integrity, and freshness for *online transaction processing* (OLTP) workloads. EnclaveDB loads the tables, indices, and metadata into the enclave memory, which is too large for efficient processing using commercially available TEEs. Furthermore, EnclaveDB has a large TCB as the query engine, transaction manager, and stored procedures are inside the enclave. As a result, the problems described for Haven and SCONE are only slightly less severe. A further downside is that all possible queries have to be known in advance.

ObliDB [30] is an Intel SGX–based encrypted database that hides the access pattern using oblivious query processing algorithms on a B^+-tree index or a linear array. The additional protection introduces a latency overhead of 200% compared to a plaintext database. Additionally, ObliDB lacks transaction management and disk persistency.

HardIDX, which is presented in Chapter 6, uses Intel SGX to protect B^+-trees. It performs equality and range searches inside the enclave, and it either loads the entire tree at once into the enclave memory or individual nodes on demand. In the second case, only a few megabytes of enclave memory are necessary and the enclave has only a few LOC. However, a B^+-tree is only one building block of an encrypted database.

Cipherbase [170] and StealthDB [29] use an FPGA and Intel SGX as trusted hardware, respectively. These approaches have the smallest TCB by putting the execution of individual, stateless operations, e.g., $<$, $>$, and $=$, into a trusted environment. The operations are executed on encrypted data and the results are passed back. Only minor changes to an application are necessary as plaintext operations are just replaced by protected operations. However, much information is leaked as an attacker learns the result of each operation (see Section 4.8.2 for more information).

7.2.2 Software-only, Encrypted Databases

Some software-only, encrypted databases, such as CryptDB [182] and Monomi [186], use PPE for efficient search. Every database functionality requires its own encryption scheme with additional storage overhead. For instance, DET [10] is used to support equality selects, and OPE [11], [12], [127], [129] allows range queries. However, the security of PPE schemes is debatable (see Section 4.4.2). For instance, Naveed *et al.* [13], Grubbs *et al.* [14], and Lacharité *et al.* [15] present attacks recovering plaintext data with a high success rate. Some encrypted dictionary types are affected by the mentioned attacks, but the data owner can freely choose a security level that fits his requirements without losing functionality. Furthermore, EncDBDB handles equality and range queries with one encryption scheme having a small performance and storage overhead.

Other approaches not using PPE have been published: Cash *et al.* [194] introduce a protocol that allows boolean query evaluation on encrypted data. Faber *et al.* [183] extend this protocol to support range queries but either leak additional information on the queried range or the result set contains false positives. Pappas *et al.* [190] evaluate encrypted bloom filters using MPC. However, in order to achieve practical efficiency, the authors propose to split the server into two non-colluding parties. Egorov *et al.* [191] present ZeroDB, a database that enables a user to perform equality and range searches with the help of B^+-trees. It uses an interactive protocol requiring many rounds and thus is not usable for network-sensitive cloud computing. EncDBDB does neither return false positives, nor does it require an additional party, nor does it need multiple rounds.

7.2.3 Searchable Encryption (SE)

We provide a detailed introduction into SE in Section 4.5. Here, we briefly compare EncDBDB's performance with SE schemes supporting range queries. Some range-searchable schemes have a linear search time [145], [147]. Others use an inverted index to achieve (amortized) polylogarithmic search time [148], [220]. Demertzis *et al.* [149] present multiple constructions that improve the constant factor of range searches. However, their best construction (without prohibitive storage cost and false positives) requires more than a second to perform a range search within 100,000 values (see Section 6.6.2). EncDBDB operates on millions of entries in milliseconds.

7.3 Design

In Section 7.1.3, we present how encrypted dictionary types are used to build EncDBDB, an encrypted, column-oriented, dictionary-encoding–based, in-memory database. Throughout this chapter, we use range queries to describe details about the nine encrypted dictionary types that EncDBDB offers. In Section 7.4, we present how EncDBDB supports other query types.

The encrypted dictionary types differ from each other in two dimensions—repetition and order of values in the dictionary—with three options each (see Table 7.2). The repetition options are the following: frequency revealing, frequency smoothing, and frequency hiding. The order options are the following: sorted lexicographically, sorted and rotated around a random offset, and unsorted. An encrypted dictionary type is defined by one option from each dimension, which leads to nine encrypted dictionary types (ED1–ED9). In the following, we present how the six options impact the encrypted dictionary types regarding security, performance, and storage efficiency. Then, we describe each encrypted dictionary type in detail.

		Order options		
		Sorted	Rotated	Unsorted
Repetition options	Frequency revealing	ED1	ED2	ED3
	Frequency smoothing	ED4	ED5	ED6
	Frequency hiding	ED7	ED8	ED9

Table 7.2: Characteristics of encrypted dictionary types.

The repetition options increase the number of repeated dictionary values from frequency revealing to frequency hiding. This affects two features of the resulting encrypted dictionary types (see Table 7.3): the security feature frequency leakage and the dictionary size $|\boldsymbol{D}|$. Note that $|\boldsymbol{D}|$ is fixed for frequency revealing and frequency hiding. For frequency smoothing, the worst-case size is $|\boldsymbol{AV}|$, but we give the average size, which depends on a configurable parameter bs^{max}.

Repetition options	Frequency leakage	Dictionary size $\lvert\boldsymbol{D}\rvert$
Frequency revealing	Full	$\lvert\mathsf{un}(\boldsymbol{C})\rvert$
Frequency smoothing	Bounded	$\sim \sum_{V \in \boldsymbol{C}} \frac{2 \cdot \lvert\mathsf{oc}(V, \boldsymbol{C})\rvert}{1 + bs^{max}}$
Frequency hiding	None	$\lvert\boldsymbol{AV}\rvert$

Table 7.3: Security feature frequency leakage and dictionary size of repetition options.

The order options affect the order of dictionary values, which determines two features of the encrypted dictionaries (see Table 7.4). First, they determine the security feature order leakage, i.e., the information an attacker with memory access can learn about the plaintext order of the encrypted values in $\boldsymbol{D}$. Second, they determine the search time combining the dictionary and attribute vector search time. The dictionary search time depends on $|\boldsymbol{D}|$ and the search algorithm, which differs for the order options. The attribute vector search time depends on the ValueIDs returned by the dictionary search, because $\boldsymbol{AV}$ has to be scanned for them.

Order options	Order leakage	Search time
Sorted	Full	$O(\log \|\boldsymbol{D}\|) + O(\|\boldsymbol{AV}\|)$
Rotated	Bounded	$O(\log \|\boldsymbol{D}\|) + O(\|\boldsymbol{AV}\|)$
Unsorted	None	$O(\|\boldsymbol{D}\|) + O(\|\boldsymbol{AV}\| \cdot \|\boldsymbol{vid}\|)$

Table 7.4: Security feature order leakage and search time of order options.

Three operations differ for the nine encrypted dictionary types:

1. Encrypted dictionary creation at the data owner.
2. Dictionary search inside the enclave at the cloud provider.
3. Attribute vector search in the untrusted realm at the cloud provider.

In the next subsections, we describe the corresponding operations in detail and denote these operations by:

1. EncDB.
2. EnclDictSearch.
3. AttrVectSearch.

As mentioned before, an encrypted dictionary type is defined by an order and a repetition option. We start by describing the frequency revealing repetition option and explain how it is combined with the three order options to instantiate ED1–ED3. Then, we do the same for the frequency smoothing repetition option and its combinations (ED4–ED6), followed by the frequency hiding repetition option and its combinations (ED7–ED9).

Throughout the following description, we always assume a closed search range to provide a concise description. However, open or half-open ranges can be handled trivially.

7.3.1 Frequency Revealing: ED1–ED3

For the frequency revealing option, the split of a column $\boldsymbol{C}$ is performed by inserting each unique column value $V \in \mathsf{un}(\boldsymbol{C})$ into $\boldsymbol{D}$ exactly once at an arbitrary position, i.e., $|\boldsymbol{D}| = |\mathsf{un}(\boldsymbol{C})| \wedge \forall V \in \mathsf{un}(\boldsymbol{C}) \colon V \in \boldsymbol{D}$. The ValueIDs in $\boldsymbol{AV}$ are set such that the split is correct according to Definition 33.

This column split provides the best compression rate possible with dictionary encoding and thus frequency revealing is the most storage efficient repetition option. However, an attacker can learn the frequency of each value $D_j \in \boldsymbol{D}$ by counting the occurrences of j in $\boldsymbol{AV}$. This is still the case if each $V \in \boldsymbol{D}$ is encrypted with AE, because the encryption does not affect the indices in $\boldsymbol{AV}$. Therefore, all encrypted dictionary types with the frequency revealing option have full frequency leakage.

ED1. For each column $\boldsymbol{C}$ that is protected with ED1, EncDB_1 performs the frequency revealing column split, sorts the values in $\boldsymbol{D}$ lexicographically, and adjusts the ValueIDs in $\boldsymbol{AV}$ such that the split is correct. Afterwards, EncDB_1 derives SK^D from the data owner's secret key SK^{DB}, the table name, and the column name, using a key derivation function. EncDB_1

then encrypts all values in $\boldsymbol{D}$ individually with AE_Enc under SK^D. The resulting dictionary containing encrypted values is denoted by $\boldsymbol{eD}$. Figure 7.3 presents an example column $\boldsymbol{C}$ and the result of ED1 before AE_Enc is performed. ED1 has full order leakage because an attacker knows the plaintext order of the encrypted values in $\boldsymbol{eD}$.

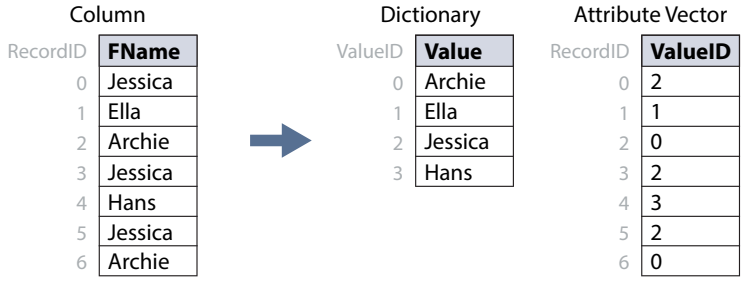

Figure 7.3: Example for ED1 before encryption.

ED1's dictionary search EnclDictSearch_1, which is executed in the enclave at the cloud provider, is presented in Algorithm 5. The function gets an encrypted range τ and an encrypted dictionary $\boldsymbol{eD}$ as input. First it derives SK^D and decrypts the start and end of the range individually. Then, one leftmost and one rightmost binary search is performed to find the dictionary indices where the searched range starts (vid_{min}) and ends (vid_{max}). All dictionary values are encrypted and stored in untrusted memory. The binary searches load the values into the enclave individually, decrypt them, and compare them with the search value. The number of load, decrypt and compare operations is logarithmic in $|\boldsymbol{D}|$. For brevity, we omit one detail in Algorithm 5, which is used in our implementation: the results of the searches, and whether a value was found, is used to handle cases in which a value is not present.

Algorithm 5 EnclDictSearch_1($\tau, \boldsymbol{eD}$)

1: $SK^D = \mathsf{DeriveKey}(SK^{DB}, columnName, tableName)$
2: $R^s = \mathsf{AE\_Dec}(SK^D, \tau^s)$, $R^e = \mathsf{AE\_Dec}(SK^D, \tau^e)$
3: $vid_{min} = \mathsf{BinarySearchLM}(\boldsymbol{eD}, R^s)$
4: $vid_{max} = \mathsf{BinarySearchRM}(\boldsymbol{eD}, R^e)$
5: **return** $\boldsymbol{vid} = [vid_{min}, vid_{max}]$

Note that only small, constant enclave memory is required for EnclDictSearch_1. This is also the case for the EnclDictSearch operations of all other encrypted dictionary types. Specifically, the required enclave memory is independent of $|\boldsymbol{D}|$.

AttrVectSearch_1 is executed in the untrusted realm at the cloud provider. It linearly scans the corresponding $\boldsymbol{AV}$, checks if the ValueIDs fall between vid_{min} and vid_{max}, and returns the matching RecordIDs $\boldsymbol{rid}$, i.e., $\boldsymbol{rid} = \{i \mid AV_i \in \boldsymbol{AV} \wedge AV_i \in [vid_{min}, vid_{max}]\}$. This operation is parallelizable with a speedup expected to be linear in the number of threads.

ED2. The idea of ED2 is to sort and randomly rotate $\boldsymbol{D}$ and it is based on an approach presented by Kerschbaum *et al.* [221]. EncDB_2 executes the frequency revealing column split, sorts the values in $\boldsymbol{D}$ lexicographically, generates a random offset *rndOffset*, and rotates $\boldsymbol{D}$ by this offset. More formally, let $\boldsymbol{D'}$ be the sorted dictionary, then $\boldsymbol{D} = (D_i \mid D_i = D'_j \wedge i = (j + rndOffset) \bmod |\boldsymbol{D'}|)$. Afterwards, EncDB_2 adjusts the ValueIDs in $\boldsymbol{AV}$ such that the split is correct. Finally, it encrypts *rndOffset* and each $V \in \boldsymbol{D}$ with AE under SK^D, resulting in *encRndOffset* and $\boldsymbol{eD}$. The order leakage of EncDB_2 is bounded, because an attacker who can observe no or a limited number of queries, does not know where the smallest and largest values are stored in $\boldsymbol{eD}$.

Figure 7.4 illustrates an example with $rndOffset = 3$ (before encryption). For instance, "Jessica" has the ValueID 2 in a sorted dictionary $\boldsymbol{D'}$. After the rotation, the ValueID is $1 = (2 + 3) \bmod 4$.

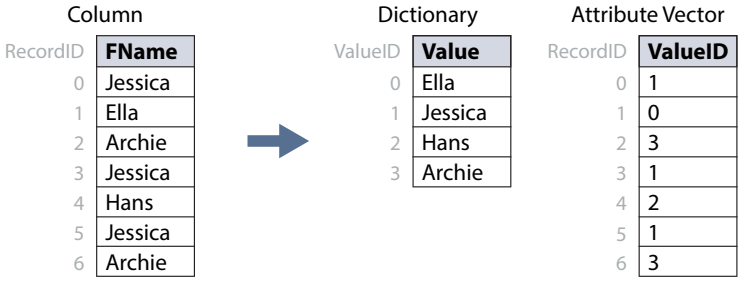

Figure 7.4: Example for ED2 with $rndOffset = 3$ and before encryption.

The processing inside the enclave (EnclDictSearch_2) is illustrated in Algorithm 6. First, EnclDictSearch_2 derives SK^D and decrypts the encrypted range τ and *encRndOffset* with it. Then, it calls a special binary search variant, which is explained in the next paragraph, to search the start and end indices of the range—vid_{min} and vid_{max}. These indices have to be processed further inside the enclave, because the positions of the indices relative to *rndOffset* define the final result of the dictionary search and *rndOffset* is sensitive. There are three possibilities: both indices are lower than *rndOffset*; both are greater than or equal to *rndOffset*; or vid_{min} is above and vid_{max} is below *rndOffset*. In the first and second case, the results are in the range $[vid_{min}, vid_{max}]$. In the third case, there are again two possibilities: vid_{min} does or does not equal $|\boldsymbol{eD}|$. In the first case, the range start was not found in $\boldsymbol{eD}$, but it is higher than the last value in it. Accordingly, all results are in the range $[0, vid_{max}]$. Otherwise, the results are split in a lower range $[0, vid_{max}]$ and an upper range $[vid_{min}, |\boldsymbol{eD}| - 1]$. We always return a dummy range if the result is only one range to simplify attribute vector search.

Algorithm 6 EnclDictSearch_2($\tau, \boldsymbol{eD}, encRndOffset$)

1: $SK^D = \mathsf{DeriveKey}(SK^{DB}, columnName, tableName)$
2: $R^s = \mathsf{AE\_Dec}(SK^D, \tau^s)$, $R^e = \mathsf{AE\_Dec}(SK^D, \tau^e)$
3: $rndOffset = \mathsf{AE\_Dec}(SK^D, encRndOffset)$
4: $vid_{min} = \mathsf{BinSearchSpecialS}(\boldsymbol{eD}, R^s, rndOffset, SK^D)$
5: $vid_{max} = \mathsf{BinSearchSpecialE}(\boldsymbol{eD}, R^e, rndOffset, SK^D)$
6: $\boldsymbol{vid} = \emptyset$
7: **if** $(vid_{min} < rndOffset \;\&\; vid_{max} < rndOffset) \;||\; (vid_{min} \geqslant rndOffset \;\&\; vid_{max} \geqslant rndOffset)$ **then**
8: $\quad \boldsymbol{vid} = \{[vid_{min}, vid_{max}], [-1, -1]\}$
9: **else if** $vid_{min} \geqslant rndOffset \;\&\; vid_{max} < rndOffset$ **then**
10: $\quad$ **if** $vid_{min} != |\boldsymbol{eD}|$ **then**
11: $\quad\quad \boldsymbol{vid} = \{[0, vid_{max}], [vid_{min}, |\boldsymbol{eD}| - 1]\}$
12: $\quad$ **else**
13: $\quad\quad \boldsymbol{vid} = \{[0, vid_{max}], [-1, -1]\}$
14: **return** $\boldsymbol{vid}$

Algorithm 7 presents the details of the special binary search with slightly different handling of the range start (BinSearchSpecialS) and end (BinSearchSpecialE). The goal is to perform a binary search that has an access pattern that is independent of *rndOffset*. A binary search that considers *rndOffset* during the data access would leak *rndOffset* in the first round, which would thwart the additional protection.

To achieve this goal, Algorithm 7 uses a string encoding operation Encode, which converts string values of a fixed maximal length to an integer representation preserving the lexicographical

data order. Each character is converted individually to an integer of fixed length and the integers are concatenated to one resulting integer. For instance, the encoding of "AB" would be 3334 and "BA" would lead to 3433. The lexicographical order is preserved by right padding the resulting integer to a fixed maximal length. In many DBMSes, the values in each column of a database have a fixed maximal length, which is fixed either implicitly by the datatype, e.g., 32 bit for `INTEGER` columns (in MySQL), or fixed explicitly with the datatype, e.g., 30 characters for `VARCHAR(30)` columns. For instance, Encode converts "AB" to the decimal 3334000000 for a `VARCHAR(5)` column.

Algorithm 7 BinSearchSpecialS/BinSearchSpecialE($\boldsymbol{eD}, sVal, rndOffset, SK^D$)

```
1: l = 0, h = |eD|
2: r = Encode(AE_Dec(SK^D, eD_0))
3: N = Encode(column maximum)
4: sVal = (Encode(sVal) − r) % N
5: while l < h do
6:     j = ⌈(l + h)/2⌉
7:     m = Encode(AE_Dec(SK^D, eD_j))
8:     cVal = (m − r)%N
9:     if (cVal < sVal) (cVal <= sVal) then
10:        l = j + 1
11:    else
12:        h = j
13: return (l) (l − 1)
```

Algorithm 7 first initializes the low and high value of the search, determines a value r by decrypting eD_0, and executes Encode on the result. Then, it performs Encode on the maximum value that fits the column, which is implicitly defined by the fixed maximal length of the column. It also executes Encode on the search value $sVal$, subtracts r from the encoded value, and takes the result modulo N. The algorithm loads all values m accessed during the search into the enclave and handles them as $sVal$. Note that 0 is a possible value for *rndOffset*, because *rndOffset* is chosen uniformly at random between 0 and $|\boldsymbol{D}| - 1$. We omit the special handling for brevity. The runtime of EnclDictSearch_2 is logarithmic in $|\boldsymbol{D}|$ and the encoding introduces only a constant factor compared to EnclDictSearch_1.

AttrVectSearch_2 linearly scans $\boldsymbol{AV}$ outside of the enclave and checks for each value $V \in \boldsymbol{AV}$ if it falls in either range that was returned by EnclDictSearch_2. Finally, AttrVectSearch_2 returns the RecordIDs $\boldsymbol{rid}$ of the matching values.

ED3. This encrypted dictionary type combines the repetition option frequency revealing and the order option unsorted. Accordingly, EncDB_3 performs the frequency revealing column split and randomly shuffles the unique values in $\boldsymbol{D}$, resulting in an unsorted dictionary. Afterwards, the ValueIDs in $\boldsymbol{AV}$ are set such that the split is correct and each $V \in \boldsymbol{D}$ is encrypted with AE_Enc under SK^D. Figure 7.5 shows an example for EncDB_3 before AE_Enc is performed. EncDB_3 trivially has no order leakage.

ED3's unsorted dictionary prevents any dictionary search with logarithmic runtime. Instead, EnclDictSearch_3 performs a linear scan over all values in $\boldsymbol{eD}$ (see Algorithm 8). EnclDictSearch_3 derives SK^D and uses it to decrypt the encrypted search range τ. The algorithm loads each $C \in \boldsymbol{eD}$ into the enclave, decrypts C, and checks if the value $V = \text{AE\_Dec}(SK^D, C)$ falls into the range $\boldsymbol{R}$. The result is a list of all matching ValueIDs $\boldsymbol{vid}$.

AttrVectSearch_3 compares each $V \in \boldsymbol{AV}$ with each $U \in \boldsymbol{vid}$ returned by EnclDictSearch_3. Thus, the runtime complexity is $O(|\boldsymbol{AV}| \cdot |\boldsymbol{vid}|)$. Integers are compared in this case,

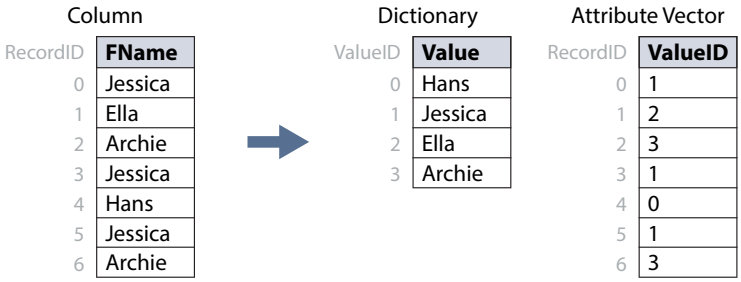

Figure 7.5: Example for ED3 before encryption.

Algorithm 8 EncIDictSearch_3($\tau, \boldsymbol{eD}$)

1: $SK^D = \mathsf{DeriveKey}(SK^{DB}, columnName, tableName)$
2: $R^s = \mathsf{AE\_Dec}(SK^D, \tau^s)$, $R^e = \mathsf{AE\_Dec}(SK^D, \tau^e)$
3: $\boldsymbol{vid} = \emptyset$
4: **for** $i = 0; i < |\boldsymbol{D}|; i{+}{+}$ **do**
5: $\quad V = \mathsf{AE\_Dec}(SK^D, eD_i)$
6: $\quad$ **if** $R^s \leq V \leq R^e$ **then**
7: $\quad\quad \boldsymbol{vid}.\mathsf{Append}(i)$
8: **return** $\boldsymbol{vid}$

which is a highly optimized operation in most CPUs. Additionally, AttrVectSearch_3 is easily parallelizable.

7.3.2 Frequency Smoothing: ED4–ED6

The main problem of the frequency revealing option is that an attacker can learn the frequency of each value $D \in \boldsymbol{D}$, even if the values are encrypted. The reason is that the underlying plaintext values are present only once with a unique ValueID. As a countermeasure, the frequency smoothing option inserts plaintext duplicates into $\boldsymbol{D}$ during the column split, bounding the frequency leakage. The foundation of this repetition option is the "Uniform Random Salt Frequencies" method presented by Pouliot *et al.* [222].

In more detail, the frequency smoothing column split executes a parameterizable and probabilistic experiment for each unique value $V \in \mathsf{un}(\boldsymbol{C})$ to determine how often V should be inserted into $\boldsymbol{D}$ (see Algorithm 9). We say that a plaintext value V is split into multiple buckets and every bucket has a specific size. The random experiment receives the number of occurrences of V in $\boldsymbol{C}$ ($|\mathsf{oc}(V, \boldsymbol{C})|$) and a bucket size maximum bs^{max}. The random size for an additional bucket is picked from the discrete uniform distribution $\mathcal{U}\{1, bs^{max}\}$ until the total size is above $|\mathsf{oc}(V, \boldsymbol{C})|$. The size of the last bucket is then set such that the total size matches $|\mathsf{oc}(V, \boldsymbol{C})|$. The experiment returns the bucket sizes $\boldsymbol{B}^{sizes}$.

Algorithm 9 getRndBucketSizes($|\mathsf{oc}(V, \boldsymbol{C})|, bs^{max}$)

1: $prevTotal = total = 0$
2: $\boldsymbol{B}^{sizes} = \emptyset$
3: **while** $total < |\mathsf{oc}(V, \boldsymbol{C})|$ **do**
4: $\quad rnd \xleftarrow{\$} [1, bs^{max}]$
5: $\quad \boldsymbol{B}^{sizes}.\mathsf{Append}(rnd)$
6: $\quad prevTotal = total$
7: $\quad total \mathrel{+}= rnd$
8: $\boldsymbol{B}^{sizes}.\mathsf{Last}() = |\mathsf{oc}(V, \boldsymbol{C})| - prevTotal$
9: **return** $\boldsymbol{B}^{sizes}$

The column split inserts $|\boldsymbol{B}^{sizes}|$ repetitions of V into $\boldsymbol{D}$. For each $i \in \mathsf{oc}(V, \boldsymbol{C})$, it randomly inserts one of the $|\boldsymbol{B}^{sizes}|$ possible ValueIDs into AV_i. Each ValueID is used exactly as often as defined by $\boldsymbol{B}^{sizes}$. As a result, the frequency leakage has a bound, because the number of occurrences of each ValueID in $\boldsymbol{AV}$ is guaranteed to be between 1 and bs^{max}.

bs^{max} can be chosen independently for each column. The selection affects $|\boldsymbol{D}|$, which determines storage efficiency, search time, and frequency leakage. For instance, a large bs^{max} leads to few repeating entries in $\boldsymbol{D}$, which slightly increases $|\boldsymbol{D}|$ compared to the frequency revealing option. This decreases the EnclDictSearch performance, because more data needs to be loaded into the enclave, more decryptions are performed, and more comparisons are necessary. The performance of AttrVectSearch also decreases, because more values have to be compared. A small bs^{max} leads to many repetitions in $\boldsymbol{D}$, which further increases $|\boldsymbol{D}|$ and the search times. Yet, it leads to a low frequency leakage bound, as each ValueID in $\boldsymbol{AV}$ is present at most bs^{max} times.

Next, we explain how the frequency smoothing column split impacts the three order options, which were introduced in detail before. We omit the discussion of order leakage as it is independent of the repetition option, and we do not explain AttrVectSearch_4–AttrVectSearch_6, because these operations are equal to AttrVectSearch_1–AttrVectSearch_3.

ED4. EncDB_4 performs the frequency smoothing column split and sorts all values in $\boldsymbol{D}$ lexicographically determining the order of repetitions randomly. Then, it adjusts the ValueIDs in $\boldsymbol{AV}$ such that the split is correct while considering how often each ValueID can be used, which is defined by $\boldsymbol{B}^{sizes}$. Finally, EncDB_4 encrypts each $V \in \boldsymbol{D}$ with AE_Enc under SK^D. Note the IND-CCA security of AE guarantees that an attacker cannot distinguish ciphertexts with an equal underlying plaintext except with negligible probability.

EnclDictSearch_4 is equal to EnclDictSearch_1, because leftmost and rightmost binary searches inherently handle repetitions. The performance penalty compared to ED1 is small, because the binary search only slows down logarithmically with a growing $|\boldsymbol{D}|$.

ED5. EncDB_5 performs the frequency smoothing column split, sorts all values in $\boldsymbol{D}$ lexicographically, rotates the ValueIDs as described in EncDB_2, sets the ValueIDs in $\boldsymbol{AV}$ such that the split is correct (considering $\boldsymbol{B}^{sizes}$), and encrypts each $V \in \boldsymbol{D}$ with AE_Enc under SK^D. Figure 7.6 shows an example for ED5 with $bs^{max} = 3$ and $rndOffset = 1$ not considering the encryption.

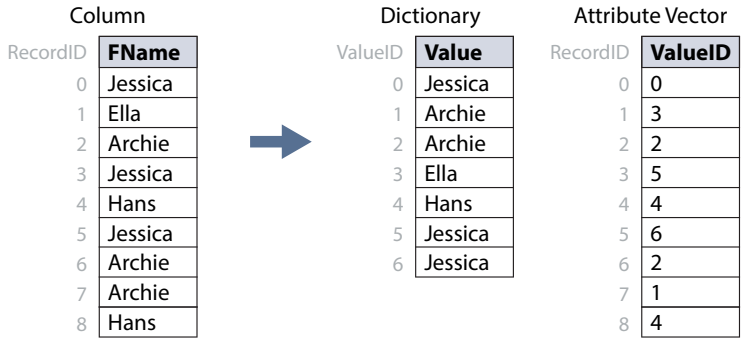

Figure 7.6: Example for ED5 with $bs^{max} = 3$ and $rndOffset = 1$ without encryption.

The special binary searches are more complex for ED5 than for ED2, because they have to handle a corner case: the plaintext value of the last and first entry in $\boldsymbol{D}$ might be equal and present more than two times (as in the example in Figure 7.6). For the same reason, EnclDictSearch_5 has to perform a more complicated postprocessing of vid_{min} and vid_{max}

compared to EnclDictSearch_2. The performance penalty compared to ED2 is small, because the binary search slows down logarithmically in $|\boldsymbol{D}|$.

ED6. For columns that are protected with ED6, EncDB_6 performs the frequency smoothing column split, randomly shuffles the values in $\boldsymbol{D}$, sets the ValueIDs in $\boldsymbol{AV}$ such that the split is correct (considering $\boldsymbol{B}^{sizes}$), and encrypts each $V \in \boldsymbol{D}$ with AE_Enc under SK^D. EnclDictSearch_6 is equal to EnclDictSearch_3, but the linear scan potentially loads, decrypts, and compares more values if $\boldsymbol{D}$ contains duplicates. If EnclDictSearch_6 returns more values compared to EnclDictSearch_3, the number of comparisons in AttrVectSearch_6 increases.

7.3.3 Frequency Hiding: ED7–ED9

Now we discuss the frequency hiding column split, which prevents frequency leakage. The idea is to add a separate entry into $\boldsymbol{D}$ for every value in $\boldsymbol{C}$, i.e., $\forall\, i \in [0, |\boldsymbol{C}| - 1]\colon D_i = C_i$. In other words, each unique value $V \in \mathsf{un}(\boldsymbol{C})$ is added $|\mathsf{oc}(V, \boldsymbol{C})|$ times into $\boldsymbol{D}$. The attribute vector is set such that the split is correct and each ValueID is used once. The resulting dictionary encoding does not provide compression anymore ($|\boldsymbol{D}| = |\boldsymbol{C}| = |\boldsymbol{AV}|$), but the frequency of every ValueID is perfectly equal, i.e., there is no frequency leakage.

ED7, ED8 and ED9. EncDB_7, EncDB_8 and EncDB_9 perform the frequency hiding column split of $\boldsymbol{C}$; sort, rotate, and shuffle $\boldsymbol{D}$, respectively; adjust the ValueIDs in $\boldsymbol{AV}$ such that the split is correct and every index in $\boldsymbol{D}$ is only used once in $\boldsymbol{AV}$; and encrypt each $V \in \boldsymbol{D}$ with AE_Enc under SK^D.

Frequency Hiding can be interpreted as a special case of frequency smoothing with a bs^{max} of 1. Therefore, the EnclDictSearch and AttrVectSearch operations are equal as described for ED4, ED5, and ED6, and the advantages and disadvantages are equivalent to the ones described for a small bs^{max}.

7.4 Extensions

In this section, we present EncDBDB extensions, which allow EncDBDB to handle joins, dynamic data, counts, aggregations, and average calculations.

7.4.1 Joins

Join operators are implemented by replacing the operations on ciphertexts with calls to the enclave. All join algorithms (i.e., hash-based, merge-sort, and nested loop joins) are compatible with all encrypted dictionary types, although optimizations are feasible but out of scope. The access pattern can be hidden using oblivious joins [223]. However, since some of the dictionaries already leak the access pattern, the additional protection only applies to some encrypted dictionaries. We present an exemplary algorithm for hash-based joins with enclave calls in Algorithm 10.

7.4.2 Dynamic Data

So far, we only discussed static data, i.e., the data owner prepares the data once and uploads it to an EncDBDB-enabled cloud provider. This is sufficient for most analytical scenarios, because bulk loading of data is often used in this context and complex, read-only queries are executed afterwards [205], [219]. For other usage scenarios, we present an approach on how EncDBDB can support dynamic data, i.e., data insertions, deletions, and updates.

Algorithm 10 Hash-based join for columns $\boldsymbol{C}^1$ and $\boldsymbol{C}^2$

```
1: for all AV ∈ AV^1 do                                  ▷ Iterate attribute vector of C^1
2:     D = ResolveDictionary(AV)
3:     h = CalculateHashInEnclave(D)
4:     HashTable.Insert(h, D)
5: for all AV ∈ AV^2 do                                  ▷ Iterate attribute vector of C^2
6:     D = ResolveDictionary(AV)
7:     h = CalculateHashInEnclave(D)
8:     T = HashTable.Get(h)
9:     for all T ∈ T do
10:        if CompareInEnclave(D, T) == true then
11:            ResultTable.Append(D, T)
```

We use a concept called *delta store* (or differential buffer) [205], [224], [225]: the database—more specifically each column—is split into a read optimized *main store* and a write optimized delta store. Both stores have a validity vector indicating whether a row is valid. New values are appended to the delta store and the corresponding rows are marked valid. For updated values, the new value is appended to the delta store, the row is marked valid, and the corresponding old row is marked invalid. Deletions are realized by an update of the validity bit. The overall state of the column is the combination of both stores. Thus, a read query becomes more complex: it is executed on both stores normally and the results are merged while checking the validity of the entries. The delta store should be kept orders of magnitude smaller than the main store to efficiently handle read queries. This is done by periodically merging the data of the delta store into the main store. Hübner *et al.* [225] describe different merging strategies.

For EncDBDB, any encrypted dictionary can be used for the main store and ED9 should be employed for the delta store. New entries can simply be appended to a column of type ED9 by re-encrypting the incoming value inside the enclave. Searches in the delta store use EnclDictSearch_9 and AttrVectSearch9. As a result, neither the data order nor the frequency is leaked during the insertion and search. A drawback of ED9 is that it has a high memory space overhead and low performance. However, the periodic merges mitigate this problem. The enclave handles the merging process as follows: First, it re-encrypts every value in $\boldsymbol{D}$. Then, columns with the rotated order option are randomly re-rotated and columns with the unsorted order option are reshuffle. The process has to be implemented in a way that does not leak the relationship between values in the old and new main store, e.g., with oblivious memory primitives [226], [227].

7.4.3 Counts, aggregations, and average calculations

For ED1–ED3 a count query is processed without any calls to the enclave by simply scanning the attribute vector, counting the occurrence of each ValueID, and returning the count together with the corresponding encrypted dictionary entry. For ED4–ED9, a count query is handled by the enclave with the following process: it performs the ValueID counting, performs a dictionary scan to merge the counts, and returns the counts together with re-encrypted dictionary entries. For all encrypted dictionaries, aggregations and average calculations are performed in the enclave with a slight deviation of the just described process: instead of merging the counts, the enclave uses the dictionary scan to calculate the aggregate or average, and returns the encrypted result.

7.5 Implementation

For our experiments, we implemented a EncDBDB prototype in C/C++ using the Intel SGX SDK in version 2.5. The prototype is based on MonetDB, an open-source, column-oriented, in-memory DBMS [210]–[212]. MonetDB focuses on read-dominated, analytical workloads and thus perfectly fits our use case. It is a commercial relational DBMS, which exploits the large main memory of modern computer systems for processing and it uses disk storage for persistency.

MonetDB uses a variant of dictionary encoding for all string columns: The attribute vector contains offsets to the dictionary, but the dictionary contains data in the order it is inserted (for non-duplicates). The dictionary does not contain duplicates if it is small (below 64 kB), and a hash table and collision lists are used to locate entries. The collision list is only used as long as the dictionary does not exceed a certain size. As a result, the dictionary might store values multiple times.

The front-end query language of MonetDB is SQL. We implemented the nine encrypted dictionary types as SQL data types in the front end and new internal data types in the back end. The encrypted dictionary types can be used in SQL create table statements like any other data type, e.g., `CREATE TABLE t1 (c1 ED7, c2 ED5, ...)`. We further split each dictionary into a dictionary head and dictionary tail. The dictionary tail contains variable length values that are encrypted with AES-128 in GCM mode. The values are stored sequentially in a random order. The dictionary head contains fixed size offsets to the dictionary tail and the values are ordered according to the selected encrypted dictionary type. This split is done to support variable length data while enabling an efficient binary search.

For dictionary searches, we pass a pointer to the encrypted dictionary into the enclave, which loads the required data from memory of the host process (see Section 3.1.2). Thus, only one context switch is necessary for each query. All operations mentioned as easily parallelizable so far run parallel in our implementation.

7.6 Evaluation

In this section, we provide security, storage, and performance evaluations of our nine encrypted dictionary types. Based on these evaluations, we conclude the section with a usage guideline regarding the different encrypted dictionary types.

7.6.1 Security Evaluation

We start the security evaluation considering the enclave size of EncDBDB. As described in Section 3.1.5, a small enclave size is crucial for security. Besides the Intel SGX SDK, the enclave of the EncDBDB prototype has only 1129 LOC. Only 412 of those LOC are written by us, the remainder is taken up by a big integer library [228] used for the dictionary search in ED2, ED5 and ED8. An enclave of this size can be efficiently verified by a user of EncDBDB.

In the remainder of this section, we discuss the security of the nine encrypted dictionary types under the attacker model defined in Section 7.1.4, i.e., an honest-but-curious attacker that targets each column independently. The attacker passively examines the processing of an encrypted dictionary $\boldsymbol{eD}$ and an attribute vector $\boldsymbol{AV}$ in multiple rounds and she knows which encrypted dictionary type is used. First, we use the following definition to describe the security of ED1–ED3 and ED7–ED9:

Definition 34 (Comparable Security)**.** *We say that the security of an encrypted dictionary type is* comparable *to a specific security scheme or definition if this security scheme or definition has*

the smallest leakage of the schemes and definitions that leak at least as much as the encrypted dictionary type.

Table 7.5 presents a summary of this evaluation on which we elaborate later. A detailed analysis of the different security definitions is beyond the scope of this dissertation as it is highly data dependent, but we reference known attacks in the same table. Afterwards, we describe the security of ED4–ED6 relative to the other encrypted dictionary types. The relation between the security provided by the different encrypted dictionary types is summarized in Figure 7.7. Finally, we discuss rerandomization.

	Frequency leakage	Order leakage	Comparable security	Known attacks
ED1	Full	Full	Ideal, deterministic ORE [132]	[14], [229]
ED2	Full	Bounded	MOPE [127]	[14], [131], [229]
ED3	Full	None	DET [10]	[13], [229]
ED7	None	Full	IND-FAOCPA [130]	[136]–[138]
ED8	None	Bounded	IND-CPA-DS [221]	[136]–[138]
ED9	None	None	RPE [148]	[136]–[138]

Table 7.5: Security of ED1–ED3 and ED7–ED9.

$$\begin{array}{ccccc} \mathsf{ED1} & \leq & \mathsf{ED2} & \leq & \mathsf{ED3} \\ \wedge\!\!| & & \wedge\!\!| & & \wedge\!\!| \\ \mathsf{ED4} & \leq & \mathsf{ED5} & \leq & \mathsf{ED6} \\ \wedge\!\!| & & \wedge\!\!| & & \wedge\!\!| \\ \mathsf{ED7} & \leq & \mathsf{ED8} & \leq & \mathsf{ED9} \end{array}$$

Figure 7.7: Relative security classification. EDX $\leq$ EDY means that EDY provides the same or better security than EDX.

ED1–ED3 and ED7–ED9. ED1's security is comparable to an ideal, deterministic variant of ORE [132]. Only a publicly known "function"—the dictionary—reveals the value order. It is ideal as neither the encrypted dictionary $\boldsymbol{eD}$ itself nor the values in it, which are encrypted with AE, leak anything but the order. It is deterministic, as equal plaintexts have the same ciphertext.

ED2's security is comparable to MOPE [127]. A column protected with ED2 only leaks the "modular" order of the values. MOPE uses deterministic OPE and ED2 uses deterministic ORE, which is more secure.

ED3's security is comparable to DET [10]. It has no order leakage, but leaks the frequency of all values.

ED7's security is comparable to IND-FAOCPA security [130]. Each ciphertext is present exactly once in $\boldsymbol{eD}$ and if a plaintext is encrypted multiple times, the assignment of each attribute vector entry to a ValueID is done with the help of a "random coin flip". Thus, the ValueIDs in $\boldsymbol{eD}$ form a randomized order (see definition by Kerschbaum [130]) of the plaintext values.

ED8's security is comparable to IND-CPA-DS security [221]. EncDB_8 and Enc as defined by Kerschbaum *et al.* [221] are different, but the security of the result is equal. Furthermore, EnclDictSearch_8 matches Search as defined by Kerschbaum *et al.* [221]. Therefore, the leakage during processing is equal.

ED9's security is comparable to the security of RPE [148]. As defined by RPE's plaintext privacy, EnclDictSearch_9 and AttrVectSearch_9 only leak the information that an entry falls

into the search range. The "predicates" of ED9 are plaintexts encrypted with AE, which provides RPE's predicate privacy.

ED4–ED6. The frequency smoothing algorithm used by ED4 makes the ciphertext frequencies close to uniform by randomly selecting a frequency between 1 and bs^{max}, independent of the plaintext frequency. As ED1 fully leaks the ciphertext frequency and ED7 hides it completely, the security of ED4 lies between the security of ED1 and ED7. ED5 is more secure than ED2 and is less secure than ED8 for the same reason. The same is true for the triple ED6, ED3 and ED9. The frequency smoothing algorithm is based on an algorithm described by Pouliot *et al.* [222] and the authors only state that the last frequency is not selected from the same distribution, which might give an advantage to an attacker. An in-depth security evaluation is an open research question.

Rerandomization. According to the data owner's sensitivity requirements, EncDBDB can use the TEE to repeat the random rotation for ED3–ED6, the random shuffle for ED7–ED9, and the random experiment for ED2, ED5, and ED8, at arbitrary points in time. Using oblivious memory primitives [226], [227], the relation between old and new encryptions is hidden.

7.6.2 Storage Evaluation

For our storage evaluation, we use a snapshot of a real-world SAP customer's *business warehouse* (BW) system. The largest columns contain 168.7 million data values. To evaluate the influence of the number of unique values to our algorithms, we search for columns having the same number of values, but different distributions. The dataset contains 30 large columns with 10.9 million values. We present the results for two extreme cases: C1 with 6.96 million unique values and C2 with 13,361.

Table 7.6 presents the storage space requirements of different variants. The plaintext file contains all plaintext values present in the column without any compression. This file is comparable to a plaintext column for which dictionary encoding is not used. The encrypted file contains every value from the plaintext file, but individually encrypted with AE, which has the same storage requirements as an encrypted column without dictionary encoding. MonetDB's storage requirements are presented as a baseline.

	Size C1	Size C2
Plaintext file	136 MB	93 MB
Encrypted file	437 MB	392 MB
MonetDB	132 MB	43 MB*
ED1/ED2/ED3	347 MB	22 MB
ED4/ED5/ED6, $bs^{max} = 100$	347 MB	56 MB
ED4/ED5/ED6, $bs^{max} = 10$	367 MB	123 MB
ED4/ED5/ED6, $bs^{max} = 2$	455 MB	331 MB
ED7/ED8/ED9	515 MB	475 MB

*Recall that MonetDB reduces some but not all duplicates (see Section 7.5).

Table 7.6: Storage size of various variants.

The size of the plaintext files decreases from C1 to C2, because the strings in these columns are 12 and 10 characters long, respectively. As expected, we see that EncDBDB requires less space if fewer unique values are present. We see that for C2 protected with ED1, ED2, or ED3, EncDBDB requires less storage space than the plaintext file, i.e., less space than a plaintext

column without dictionary encoding. We also see a further expected behavior: a smaller bs^{max} increases the required storage space as more duplicates are stored.
Note that the encrypted dictionaries are stored outside of the enclave and individual values are loaded and decrypted. Hence, Intel SGX's restricted EPC memory does not constitute a limitation for EncDBDB.

7.6.3 Performance Evaluation

For the performance evaluation, we use the same columns introduced in the storage evaluation. Besides the original columns, which we call *full datasets*, we sample datasets from 1 to 10 million records using the distribution and values of the original columns.

MonetDB is used as one baseline measurement in our experiments to compare ourselves against a commercial plaintext DBMS. Additionally, we implement *PlainDBDB*—a plaintext variant of EncDBDB. PlainDBDB uses the same algorithms as EncDBDB, but the dictionaries are plaintext and the algorithms are processed without an enclave. We use PlainDBDB as a second baseline to evaluate the performance overhead of encryption and Intel SGX.

All experiments are performed with the confidential computing offering provided by Microsoft Azure [172]. We use a DC4s machine with 4 Intel SGX–enabled vCPUs of an Intel Xeon E-2176G @ 3.70 GHz and 16 GB RAM. All presented latencies measure the processing time spent at the server excluding any network delay or processing at the proxy or user. Our protocol runs in one round and only encrypts the values in the query. Thus, the communication and latency overhead compared to any database in the cloud is negligible.

We use the term *range size* (RS) to describe how many consecutive unique values from the dataset are searched in a range query, i.e., if $\mathsf{sorted}(\mathsf{un}(\boldsymbol{C})) = (V_0, \ldots, V_{|\mathsf{un}(\boldsymbol{C})|-1})$ is a sorted list of all unique values in $\boldsymbol{C}$, then RS defines the search range $\boldsymbol{R} = [V_i, V_{i+RS-1}]$ for $i \in [0, |\mathsf{un}(\boldsymbol{C})| - RS]$. For every dataset and encrypted dictionary type, we perform 500 random range queries with range sizes 2 and 100. The same random range queries are executed for MonetDB, PlainDBDB, and EncDBDB. Note that the number of result rows returned by the server is greater than RS if a value in the search range is present multiple times in the column (see Figure 7.8). For instance, 65,067 values are returned on average for the full dataset of C2 and $RS = 100$.

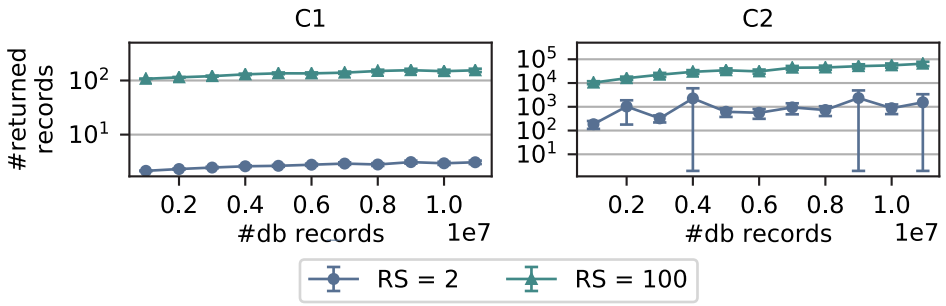

Figure 7.8: Average number of results returned by 500 random range queries for columns C1 and C2 (95% confidence interval; note that logarithmic y-axis distorts error bars).

ED1. The first and fourth column in Figure 7.9a present the latencies of ED1 for C1 and C2 and the range sizes 2 and 100. We highlight three observations from these plots. First, EncDBDB and PlainDBDB outperform MonetDB for both range sizes at both columns. The main reason is that MonetDB's attribute vector search performs a linear number of string comparisons. In contrast, EncDBDB and PlainDBDB require only a logarithmic number of string comparisons in the dictionary search and a linear number of integer comparisons in

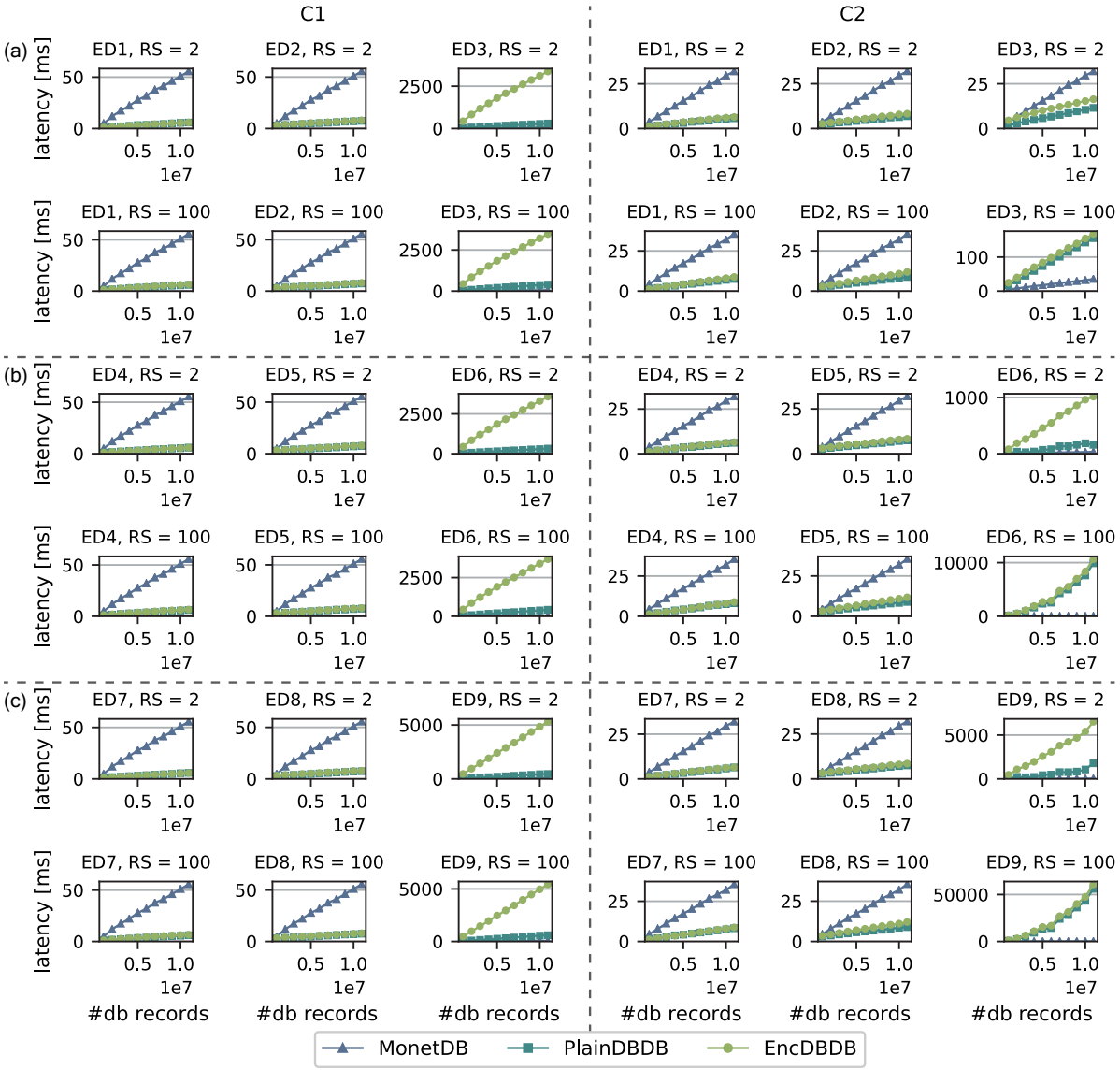

Figure 7.9: Average latencies of 500 random range queries for columns C1 and C2, which are protected by (a) ED1–ED3, (b) ED4–ED6, and (c) ED7–ED9 (95% confidence interval not big enough to be visible).

the attribute vector search. Second, EncDBDB slows down if a column with equal size has fewer unique values: the average latencies increase from 6.55 ms at C1 to 8.79 ms at C2 for the full dataset and $RS = 100$. This seems counterintuitive, because fewer unique values result in a smaller dictionary size $|\boldsymbol{D}|$ improving the dictionary search performance. However, only logarithmically fewer decryptions and string comparisons are necessary in the dictionary search, but many results are returned by the attribute vector search (see Figure 7.8). As a result, the DBMS has to spend more time for tuple reconstruction, i.e., to build the result set based on the found RecordIDs and the dictionary. Third, encryption is cheap: the average latency overhead of EncDBDB compared to PlainDBDB is 0.36 ms (8.9 %). The overhead is minor for two reasons: (1) as explained in the implementation section, only one context switch per column is required, which is negligible in the overall latency and (2) hardware-supported AES-GCM encryption is used.

ED2. The second and fifth column in Figure 7.9a present the latencies of ED2. The main observation is that the latency of EncDBDB and PlainDBDB is almost equal to the latency of ED1 for the two columns. The only difference between ED1 and ED2 is that ED2 uses a special binary search and post-processing of the resulting ValueIDs to handle the random rotation, which introduces only a minor overhead. In fact, the average latency overhead from ED1 to ED2 is 1.88 ms for EncDBDB.

ED3. The third and sixth column in Figure 7.9a show the latencies of ED3. We observe that the average latencies of PlainDBDB and EncDBDB, and their relative latency differences, severely depend on the number of unique values and the range size RS. C2 has a smaller $|\boldsymbol{D}|$ than C1, which decreases the latency of the linear dictionary search and therefore the average latency of the query execution. Additionally, a smaller $|\boldsymbol{D}|$ decreases the number of necessary decryptions for EncDBDB and therefore the relative latency difference between PlainDBDB and EncDBDB.

ED4, ED5, ED6. Figure 7.9b presents the latency plots for ED4–ED6. The latencies of MonetDB obviously do not change. In the following, we focus on EncDBDB discussing the latencies for ED4–ED6 compared to ED1–ED3. $|\boldsymbol{D}|$ is larger for ED4–ED6, because the frequency smoothing algorithm adds duplicates to $\boldsymbol{D}$ ($bs^{max} = 10$ in our experiments). For ED4 and ED5, $|\boldsymbol{D}|$ influences the latency only logarithmically. Compared to ED1 and ED2, the average overheads are only 0.002 ms and 0.11 ms, respectively. At ED6, the dictionary search might return more than x ValueIDs for the range size x as $\boldsymbol{eD}$ contains duplicate plaintexts. Every returned value has to be compared to each attribute vector entry. This increases the average latencies for the full dataset at $RS = 100$ to 3.59 s and 10.64 s for C1 and C2.

ED7, ED8, ED9. Figure 7.9c presents the latency plots for ED7–ED9. We again focus on EncDBDB's latency in ED7–ED9 compared to ED1–ED3. Compared to ED1 and ED2, the average overheads of ED7 and ED8 are 0.01 ms and 0.23 ms, respectively. For the full dataset at $RS = 100$, the average latencies of ED9 increase to 5.43 s and 60.82 s for C1 and C2, respectively.

7.6.4 Usage Guideline

The data owner can select an arbitrary encrypted dictionary type per column, according to the desired sensitivity. If plaintext is not an option, but the weakest security level is acceptable, ED1 can be used. It has a small storage size and it is almost as fast as PlainDBDB, even with different range sizes and unique value amounts. If order leakage should be reduced and a minor performance overhead is acceptable, ED2 is preferable over ED1. If order leakage is unacceptable, a column contains few unique values, and RS is small, ED3 has a practical overhead. For instance, EncDBDB's average latency overhead from ED1 to ED3 for C2 and $RS = 2$ is 6.87 ms. If the frequency leakage should be bounded, ED5 can be used with a minor performance and storage overhead compared to ED2. In many cases, ED5 is the best security, latency, and storage trade-off among our encrypted dictionary types. If security and latency are critical, but not storage size, ED8 is the most favorable encrypted dictionary. If security is the main objective, ED9 should be used.

7.7 Summary

In this chapter, we used TEE-protected dictionaries to build a secure, efficient, outsourced, column-oriented, in-memory database—EncDBDB. This system supports analytic queries on large datasets providing nine encrypted dictionary types with distinct security, performance,

and storage efficiency trade-offs. Range queries over columns with almost 11 million encrypted rows require less than 13 ms (on average), even without frequency leakage and bounded order leakage. If some frequency leakage is acceptable, the compressed encrypted data requires less space than a plaintext column. Moreover, the TCB of EncDBDB consists only of 1129 LOC exposing only a small attack surface. With those features, EncDBDB is ideally suited for an entity that wants to outsource its database containing sensitive data to an untrusted cloud environment.

8
Protected File System: SeGShare

In this chapter, we examine the secure, outsourced, TEE-based processing of a *file system*. A file system contains files and directories, whereby a directory is a collection of files and/or further directories. We choose to focus on the sharing capabilities of file systems, as file sharing between users is a common scenario across many applications. One option is to distribute the files to each group member individually. A better option is to use a shared file system to store files and manage access control.

Using a TEE-protected file system, we design *SeGShare*—a secure, efficient, outsourced system for group file sharing supporting large and dynamic groups. All sensitive processing steps are done inside an enclave, which is deployed at an untrusted cloud provider. Via authentication tokens, users authenticate themselves to the enclave and establish a secure channel with it, e.g., using TLS. This channel is used for all subsequent communication. On every user access, the enclave checks encrypted access control policies to enforce read and/or write access on files and directories. Users can upload arbitrarily large files through the secure channel directly to the enclave. If the upload is granted, the enclave encrypts the files with an AE scheme (see Section 2.2.5) under a random key and stores the encrypted files in the untrusted environment. On each granted file request, the file is decrypted inside the enclave and sent to the user over the secure channel. SeGShare separates authentication and authorization using identity information in authentication tokens. As long as the identity information is preserved, a user's token can be replaced and a user can use different tokens for multiple devices—without requiring server-side file changes or re-encryptions. Furthermore, immediate permission and membership revocations only require a minor, inexpensive modification of an encrypted file. Among other features, SeGShare protects the confidentiality and integrity of all data and administration files; supports data deduplication; and mitigates rollback attacks. A comprehensive list of SeGShare's features is presented in Table 8.2.

We agree with the authors of A-SKY [230], who state that "given the memory and computational limitations of SGX enclaves (e.g., trusted computing base (TCB) size, trusted/untrusted transition latency), it is far from trivial to develop such a [file and access control] proxy service able to scale and sustain a high data throughput, considering dynamic access control operations." The key to achieve a high throughput under dynamic groups is that SeGShare does not require complex cryptographic operations on permission or membership changes. Besides that, we build an efficient, Intel SGX–enabled TLS stack; use switchless enclave calls for all network and file traffic; and the enclave requires only a small, constant size buffer for each request.

In the evaluation section of this chapter, we examine SeGShare's security, storage, and performance. Regarding security, we focus on the end-to-end protection of user files, which is achieved by multiple design decisions of SeGShare. Based on a SeGShare prototype, we show how small the storage overhead of encrypted files is compared to plaintext files. Additionally, we use the prototype to present five performance experiments measuring the latency of different operations, e.g., file uploads, file downloads, and permission revocations.

Some cloud providers, e.g., MEGA [231] and Sync.com [232], offer secure group file sharing systems. They use *hybrid encryption* (HE) [233] to enforce access control, i.e., a file is encrypted with a unique, symmetric *file key*, and the file key is encrypted with the public key of each user

that should have access. Besides HE, many cryptographic schemes have been proposed [160], [161], [234]–[239] to improve access control policies regarding, e.g., key distribution, the number of keys, and expressiveness. Various papers use these schemes to build cryptographically protected file sharing systems [233], [237], [240]–[243]. The main drawback of such systems is that users gain plaintext access to the file key. To achieve immediate permission revocation, it is necessary to re-encrypt the corresponding file with a new key. Depending on the scheme, this involves expensive cryptographic operations and the new key has to be distributed to many users. The problem is more severe on immediate group membership revocation, as many files require re-encryption. This becomes a critical problem if membership operations occur frequently [244]. Alternatively, a TEE can be used to build secure group file sharing systems. Some approaches use cryptographic access control schemes and thus also suffer from users' access to plaintext file keys [230], [245]. Other approaches propose a client-side enclave, which is a severe drawback due to the heterogeneity of end-user devices [246]. The approach most comparable to SeGShare is Pesos [247], but Pesos does not offer multiple important features, e.g., deduplication of encrypted files, rollback protection, and confidentiality of group memberships.

The contributions of this chapter are the following:

- A secure, efficient, outsourced, TEE-based group file sharing system supporting large and dynamic groups.
- The first file sharing system combining confidentiality and integrity of all data and administration files; immediate revocations without expensive re-encryption; data deduplication; rollback protection; and separation of authentication and authorization.
- A latency average of 2.39 s and 2.17 s to upload and download a 200 MB file—faster than Apache WebDAV serving plaintext files in the same setup. Latency under 170 ms for membership and permission operations, independent of file sizes, stored files in total, number of group members, number of user permissions, and groups sharing a file.
- A storage overhead of only 1.06% for an encrypted file shared with more than 1000 groups containing 200 MB plaintext data.
- An Intel SGX–enabled, optimized TLS implementation.
- An enclave with only 8441 LOC, reducing the potential for security-relevant implementation errors, unintended leakages, hidden malware, and side-channel leakages.

The remainder of this chapter is structured as follows: In Section 8.1, we present a file system model, introduce SeGShare's access control model, specify SeGShare's design objectives, and state the attacker model. Afterwards, we discuss to which extent related file sharing systems fulfill the design objectives in Section 8.2. In Section 8.3, we present SeGShare's design in detail, which fulfills most of the design objectives. The remaining design objectives are fulfilled by SeGShare extensions introduced in Section 8.4. Security and performance-critical implementation details of our SeGShare prototype are discussed in Section 8.5. Based on the SeGShare design and the prototype, we provide in-depth security, storage, and performance evaluations in Section 8.6. Finally, Section 8.7 concludes this chapter with a summary.

8.1 Design Considerations

In this section, we first describe the data structure under consideration, a file system. Then, we give an overview of SeGShare introducing its access control model, illustrating its features, and deducing a set of functional, performance, and security objectives. Finally, we describe the assumed attacker model.

8.1.1 Data Structure: File System

We use a generic file system model that applies to various operating systems (cf. the file system model by Tanenbaum *et al.* [248]). A *file system* ($\boldsymbol{FS}$) is composed of *files* ($\boldsymbol{F}^C$) and *directories* ($\boldsymbol{F}^D$). We denote the former *content files* and the latter *directory files* as both are stored in files. Each $F^C \in \boldsymbol{F}^C$ contains a linear array of bytes that can be read and written. Each $F^D \in \boldsymbol{F}^D$ is a collection of files and/or further directories, and it stores a list of all its children. The directories form a tree with a *root directory file* ($F^{D^{root}}$) at the root of the tree. The *parent directory* of each $F \in \boldsymbol{FS}$ is specified by its parent in the tree.

Each $F^D \in \boldsymbol{F}^D$ has a *directory name*. The directory name of $F^{D^{root}}$ is defined as "/", and all other directory names are flexible excluding the character "/". Furthermore, Each F^D has a *path* that is specified by its location in the directory tree hierarchy: the path is the concatenation of all directory names in the tree from $F^{D^{root}}$ to F^D delimited and concluded by "/". Each $F^C \in \boldsymbol{F}^C$ has a *filename*, and F^C's path is the concatenation of the path of its parent directory and its filename.

8.1.2 System: SeGShare

Based on the file system described in the last section, we build SeGShare—a secure, efficient, outsourced system for group file sharing using a TEE. In the following, we provide an overview of SeGShare, define SeGShare's objectives, and introduce a formal access control model. In the design and extension sections (Section 8.3 and Section 8.4), we describe how SeGShare achieves the defined objectives in detail.

SeGShare involves three entities: a trusted *file system owner* (FSO), multiple trusted *users* ($\boldsymbol{U}$), and an untrusted *cloud provider* with a trusted (Intel SGX) enclave. The FSO has a relation to the users $\boldsymbol{U}$, e.g., the FSO is a company and the users are employees of this company. The users want to share files among each other via a file-sharing system hosted at the cloud provider.

The FSO has an *authentication service*, which provides an *authentication token* with identity information to all users. To use EncDBDB, each user only has to store its authentication token. Each user uses its token while establishing a secure channel with the enclave running at the cloud provider. Without any special hardware, users use the established secure channel for the following requests: create/update/move/download/remove files; create/list/move/remove directories; set file/directory permissions for an individual user or a group; create groups; and change group memberships. None of these requests requires interaction with other users. For each request, authentication is done with the identity information contained in the authentication token and authorization is done based on permissions stored for the authenticated identity at the cloud server. This separation of authentication and authorization allows the replacement of a user's authentication token or the use of different authentication tokens for multiple devices as long as the identity information is preserved. W.l.o.g., we use a *certificate authority* (CA) as authentication service and certificates as authentication tokens throughout this chapter.

Table 8.1 presents an overview of SeGShare's access control model. A user $U \in \boldsymbol{U}$ is assigned to one group $G \in \boldsymbol{G}$ or multiple groups. Additionally, each user U is part of its *default group* G^U, i.e., a group that only contains U. Each group G has at least one *group owner* (GO), which initially is the user U adding the first member to G. A GO can change *group memberships* ($\boldsymbol{R}^G$) and extend the *group ownership* ($\boldsymbol{R}^{GO}$) to other groups. Every $F \in \boldsymbol{FS}$ has at least one *file owner* (FO), which initially is the user uploading a file or creating a directory. For any file F and group G, the FO can extend the *file ownership* ($\boldsymbol{R}^{FO}$) and set *file permissions* ($\boldsymbol{R}^P$). The permission can either be a combination of read (P^r) and write permissions (P^w), or

access can be denied (P^{deny}). As a result, a user's permissions depend on the permissions of all groups he is a member of. The main benefit of group-based permission definitions is that a membership update is sufficient to provide or revoke a user's access to many files instead of changing the permissions of all affected files individually. Each file owner can define that a file $F \in \boldsymbol{FS}$ should inherit permissions from its parent ($\boldsymbol{R}^I$). This enables, for example, central permissions management of multiple files: create a directory, set the desired permissions for the directory, add files to the directory, and define that the files should inherit permissions.

Element	Description
$\boldsymbol{U}$	Set of individual users U
$\boldsymbol{G}$	Set of individual groups G; each user U has a default group G^U
$\boldsymbol{P}$	Set of individual permissions $P \in \{P^r, P^w, P^{deny}\}$
$\boldsymbol{F}^C$	Set of stored individual content files F^C
$\boldsymbol{F}^D$	Set of stored individual directory files F^D
$\boldsymbol{FS}$	File system $\boldsymbol{FS} = \boldsymbol{F}^C \cup \boldsymbol{F}^D$
$\boldsymbol{R}^G \subset \boldsymbol{U} \times \boldsymbol{G}$	$(U, G) \in \boldsymbol{R}^G$: user U is member of group G
$\boldsymbol{R}^{GO} \subset \boldsymbol{G} \times \boldsymbol{G}$	$(G, G') \in \boldsymbol{R}^{GO}$: group G owns group G'
$\boldsymbol{R}^{FO} \subset \boldsymbol{G} \times \boldsymbol{FS}$	$(G, F) \in \boldsymbol{R}^{FO}$: group G owns file F
$\boldsymbol{R}^P \subset \boldsymbol{P} \times \boldsymbol{G} \times \boldsymbol{FS}$	$(P, G, F) \in \boldsymbol{R}^P$: group G has permission P for file F
$\boldsymbol{R}^I \subset \boldsymbol{FS}$	$F \in \boldsymbol{R}^I$: file F inherits permissions from its parent

Table 8.1: SeGShare's access control model.

We also expect the following features from a secure, efficient, outsourced file sharing system: (1) *Immediate revocation*, i.e., file permission and membership updates, especially revocations, are enforced instantly without time-consuming re-encryption of files $F \in \boldsymbol{FS}$. (2) A constant number of ciphertexts for each $F \in \boldsymbol{FS}$, independent of permissions and group memberships. (3) Confidentiality and integrity protection of all content files, the file system structure, permissions, existing groups, and group memberships. (4) Storage space reduction by file deduplication and use of the same encrypted files for different groups. (5) Rollback protection for individual files and the whole file system.

Overall, SeGShare fulfills the objectives listed in Table 8.2.

8.1.3 Attacker Model

All users trust the CA and know its public key, which is the case in many corporate environments. The CA securely creates certificates for users and securely provisions the certificates to the users. At the cloud provider, we assume a malicious attacker, i.e., an attacker that does not need to follow the protocol and tries to gain as much information as possible. Only the enclave is protected by a TEE and it supports the capabilities listed in Table 3.1. The code is assumed not to have intentional data leakage and it contains a hard-coded copy of the CA's public key. All other software is controlled by the attacker. As a result, she can monitor and/or change data on disk or in memory; rollback individual files or the whole file system; send arbitrary requests to the enclave; view all network communications; and monitor communication between the untrusted and trusted software part. An attacker controlling multiple users should only have permissions according to the union of permissions of the individual controlled users. We do not protect the number of files, the file sizes, and the file access pattern.

As we show in Section 3.2, Intel SGX is vulnerable to various attacks and multiple mitigations are known. We consider attacks on Intel SGX and their mitigations an orthogonal problem and do not consider them in this chapter. However, SeGShare has small enclave size and therefore the mitigations are straightforward to integrate. Hardware and DoS attacks are out of scope.

Obj.	Description
F1	File sharing with individual users / groups
F2	Dynamic permissions / group memberships
F3	Users set permissions
F4	Separate read and write permissions
F5	Users (and administrators) do not need special hardware
F6	Non-interactive permission / membership updates
F7	Multiple file owners / group owners
F8	Separation of authentication and authorization
F9	Deduplication of encrypted files
F10	Permissions can be inherited from parent directory
P1	Constant user storage
P2	Group-based permission definition
P3	File permission / group membership revocations do not require re-encryption of content or directory files
P4	Constant number of ciphertexts for content and directory files
P5	Different groups can access the same encrypted file
S1	Protect confidentiality of content files / file system structure / permissions / existing groups / group memberships
S2	Protect integrity of content files / file system structure / permissions / existing groups /group memberships
S3	End-to-end protection of user files
S4	Immediate revocation
S5	Protection against rollback of individual files / whole file system

Table 8.2: Expected functional (Fx), performance (Px) and security objectives (Sx). Sub-objectives are separated by "/".

8.2 Related Work

In this section, we discuss to which extent related file sharing systems fulfill the design objectives defined in Table 8.2. We distinguish between pure cryptographically protected and TEE-supported file sharing systems. We begin with an introduction to cryptographic access control mechanisms, because some file sharing systems are based on these mechanisms. For the related systems, Table 8.3 provides an overview of the fulfilled objectives and (if applicable) on which access control mechanism the systems are based on.

8.2.1 Cryptographic Access Control Mechanisms

A simple access control mechanism is *hybrid encryption* (HE): a file is encrypted with a unique, symmetric *file key*, and the file key is encrypted with the public key of each user that should have access. HE requires public-key management, e.g., a *public key infrastructure* (PKI), to establish a trusted connection between users and public keys.

Identity-based encryption (IBE) [234], [235], [249] allows to use arbitrary strings as public key for each user. By replacing HE's public key primitives with IBE, public key management is not necessary anymore.

Attribute-based encryption (ABE) [160], [161], [250] enables fine-grained access control by defining a set of attributes for users and files. Files can only be decrypted if a defined number of attributes match [160], the policy defined in the user's secret key matches the ciphertext's attribute [161], or the policy defined in the ciphertext matches the user's attributes [250].

The idea of *broadcast encryption* (BE) [236], [237], [251] is that a broadcaster encrypts messages, sends them via a broadcast channel, and only a permitted subset of users is able to

System	Based on	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
Pure Cryptographically Protected File Sharing Systems											
Goh *et al.* [233]	HE	●/–	●/–	●	◐	●	●/–	○/–	○	○	○
Kallahalla *et al.* [240]	HE	●/–	●/–	●	●	●	○/–	○/–	○	○	○
Boneh *et al.* [237]	BE	●/–	●/–	●	○	●	●/–	○/–	○	○	○
Garrison *et al.* [244]	IBE, ABE	●/●	●/●	○	◐	●	●/●	○/○	○	○	○
Popa *et al.* [241]	BE	●/–	●/–	○	◐	●	●/–	○/–	○	○	○
Li *et al.* [243]	ABE	●/–	●/–	●	○	●	●/–	○/–	○	●	○
TEE-Supported File Sharing Systems											
Contiu *et al.* [230]	HE	●/●	●/●	○	●	●	●/●	○/○	◐	○	○
Contiu *et al.* [245]	IBBE	●/●	●/●	○	○	○	●/●	○/○	○	○	○
Djoko *et al.* [246]		●/●	●/●	●	●	○	●/○	○/○	●	○	○
Krahn *et al.* [247]		●/●	●/○	●	●	●	●/○	●/○	●	○	○
SeGShare		●/●	●/●	●	●	●	●/●	●/●	●	●	●

System	P1	P2	P3	P4	P5	S1	S2	S3	S4	S5
Pure Cryptographically Protected File Sharing Systems										
Goh *et al.* [233]	●	○	○/–	○	–	●/○/○/–/–	●/●/●/–/–	●	●	◐/○
Kallahalla *et al.* [240]	○	○	○/–	○	–	●/○/●/–/–	●/○/–/–/–	●	○	○/○
Boneh *et al.* [237]	●	○	○/–	●	–	●/○/○/–/–	○/○/○/–/–	●	●	○/○
Garrison *et al.* [244]	●	●	○/○	○	●	●/○/○/○/○	●/○/●/●/●	●	○	○/○
Popa *et al.* [241]	●	○	○/–	●	–	●/–/○/–/–	●/–/●/–/–	●	○	◐/–
Li *et al.* [243]	●	○	○/–	●	–	●/○/○/–/–	◐/○/○/–/–	●	●	○/○
TEE-Supported File Sharing Systems										
Contiu *et al.* [230]	●	●	○/○	○	○	●/○/○/●/●	●/○/●/●/●	●	○	○/○
Contiu *et al.* [245]	●	●	○/○	○	○	●/○/○/○/○	○/○/○/○/○	●	○	○/○
Djoko *et al.* [246]	○	●	●/●	●	○	●/●/●/○/●	●/●/●/○/●	●	●	◐/○
Krahn *et al.* [247]	●	●	●/●	●	●	●/–/●/○/○	●/–/●/●/●	●	●	○/○
SeGShare	●	●	●/●	●	●	●/●/●/●/●	●/●/●/●/●	●	●	●/●

Table 8.3: Classification of SeGShare and related work based on objectives defined in Table 8.2. The symbols represent that an objective is supported (●), partially supported (◐), not supported (○), or not part of the design (–). Note that some objectives in Table 8.2 have multiple sub-objectives separated by "/", which are also used in this table.

decrypt the messages. BE can be used for file sharing by considering the files as messages and the file system as a broadcast channel. BE schemes have various trade-offs regarding private key, public key, and ciphertext size, but no scheme is constant in all sizes.

Identity-based broadcast encryption (IBBE) [238], [239], [252] is a combination of IBE and BE. Messages are encrypted under a public key for receivers that are identified by an arbitrary string, and receivers can decrypt messages with their private keys if their identity is part of the receiver set. As with BE schemes, IBBE schemes are not constant in all keys.

8.2.2 File Sharing Systems

All pure cryptographically protected file sharing systems use the just presented cryptographic access control mechanisms and enrich them with, e.g., key regression [240], integrity proofs [241], and data deduplication [243]. Some TEE-based file sharing systems also use the cryptographic access control mechanisms to design an anonymous file sharing system [230] or an IBBE scheme

with reduced encryption complexity [245]. NEXUS [246], Pesos [247], and SeGShare are not based on the cryptographic access control mechanisms.

All file sharing systems that are based on a cryptographic access control mechanism use HE or a combination of HE and IBE, ABE, BE, or IBBE. This, e.g., allows to remove the public-key management, reduces the number of keys, and/or increase the expressiveness of access control policies. However, permitted users gain plaintext access to the file key on each file access. Therefore, the following process has to be executed to enforce immediate permission revocation: a new file key is generated, the file is re-encrypted with the new key, and the new key is encrypted for each user or group still having access. On membership revocation, the just mentioned process has to be performed for every file the group has access to. Objective P3 is not fulfilled by those file sharing systems.

Additionally, Garrison *et al.* [244] state that (1) most IBE schemes are pairing-based, which is an order of magnitude slower than public-key encryption used at HE, (2) ABE incurs substantially higher costs than IBE, even for simple access control policies, (3) existing schemes for proxy re-encryption [253], [254] do not solve the problem, and (4) cryptographic access controls lead to prohibitive computational cost for practical, dynamic workloads.

Only Pesos [247] does support multiple file owners and no system supports multiple group owners (F7). Only NEXUS [246] and Pesos [247] separate authentication and authorization (F8). Deduplication of encrypted files (F9) is only supported by REED [243], and no system supports permission inheritance (F10). Most cryptographically protected file sharing systems do not support group-based permission definition (P2). Therefore, each objective related to groups is not part of their design. Only the approach from Garrison *et al.* [244] and Pesos [247] support that different groups can access the same encrypted file (P5), which can significantly reduce the required storage for files shared with different groups. Only NEXUS [246] protects the confidentiality and integrity of the file system (S1 and S2). Half of the systems perform immediate revocation (S4), and all others propose lazy revocation, i.e., re-encryption is deferred until the next file update. This opens a window of opportunity for security breaches, as a revoked user can still access all files that are not updated. No related work provides a rollback protection for individual files and the whole file system, which is enforced at every point in time (S5).

8.3 Design

In this section, we discuss SeGShare's design, which fulfills most objectives presented in Table 8.2. The remaining objectives are fulfilled by extensions discussed in Section 8.4. Our following discussion uses SeGShare's components, which are illustrated in Figure 8.1. We start with the setup phase in which trust between users and the enclave is established, followed by the runtime phase in which user requests are processed.

8.3.1 Setup Phase

The setup phase of SeGShare establishes bilateral trust between each user $U \in \boldsymbol{U}$ and the enclave running at the cloud provider. This phase is only executed once, and the established trust is the basis for the end-to-end security of user files (see obj. S3 in Table 8.3).

Establish user trust in the enclave. The CA's certificate service component connects to the untrusted certification component, one external interface of SeGShare, to perform remote attestation of the enclave (see Section 3.1.3). The CA's public key is hard-coded into the enclave. Thus, if the CA receives the expected measurement, it is assured to communicate with an enclave that was built specifically for this CA. During remote attestation, the CA

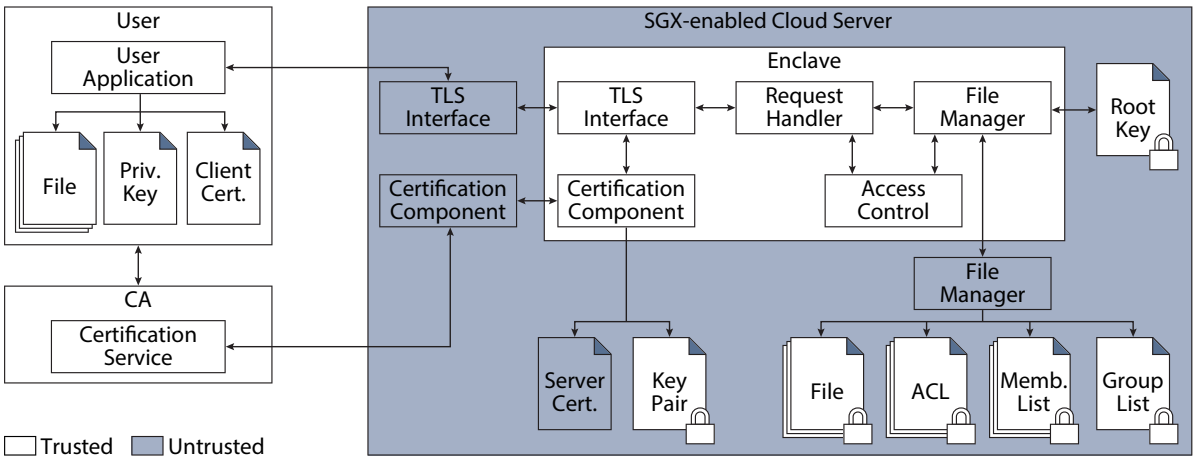

Figure 8.1: SeGShare architecture.

establishes a secure channel that ends at the trusted certification component. This channel is used for all messages exchanged in the following process:

1. The CA requests a *certificate signing request* (CSR) from the enclave.
2. The enclave generates a temporary key pair and provides the CA with a CSR containing the public key of the temporary key pair.
3. The CA uses the received CSR to generate a server certificate. Afterwards, the CA signs this certificate with its private key and provides the certificate to the enclave.
4. The enclave checks the certificate's validity using the hard-coded public key of the CA. On success, the enclave persists the server certificate in untrusted memory, seals the key pair (see Section 3.1.4), and triggers the trusted TLS interface to update its server certificate.

The CA can request a new CSR and subsequently replace the server certificate at any time.

During runtime, the users receive the server certificate on every connection. As the CA checks the validity of the enclave and the users trust the CA, the users only have to verify the server certificate with the CA's public key to be sure that they communicate with a trusted SeGShare enclave. Notably, the user does not need to perform remote attestation.

Establish enclave trust in users. For each user $U \in \boldsymbol{U}$, the CA validates U's identity and securely provides a client certificate to U. This certificate contains identity information, e.g., a user ID, a mail address, and/or a full name. A user U can check that the certificate is signed by the trusted CA as U knows CA's public key. During the TLS handshake, U's user application presents U's client certificate to the enclave, which validates the certificate using the CA's public key. On success, the enclave can be sure that it communicates with a valid user of the system.

8.3.2 Runtime Phase

Each request starts and each response ends at a user application. Therefore, we introduce the user application first and then explain which parts of the processing are done by the other components.

User Application. The user application links the users' local file systems to the remote file system at the cloud provider. For this link, it establishes a connection to SeGShare's second external interface, the untrusted TLS interface. This interface is used to establish a secure TLS

connection ending at the trusted TLS interface. Further details about the two TLS interfaces are discussed in the corresponding component description.

After a TLS handshake, the user application can send requests to the enclave. We only discuss the following requests in detail: create a directory; create/update a file; get file content or directory listing; set file/directory permission for a group; and add/remove a user to/from a group. Remember that each user is part of its default group; thus, permission requests also apply for individual users. A request defining that a file should inherit its permissions is described in Section 8.4.2. We do not discuss the following requests for brevity, but their implementation is straightforward: remove file/directory; move file/directory; update ownership of file/directory; update group ownership; and delete group.

Notably, the user application does not require any special hardware (see obj. F5), and it only needs to store a client certificate and the corresponding private key, independent of the number of outsourced files, permissions or group memberships (see obj. P1). SeGShare has low hard- and software requirements at the user making it widely applicable.

TLS Interface. The TLS interface is partitioned into an untrusted and trusted part (inside the enclave). The untrusted TLS interface terminates the network connection (e.g., TCP), because the enclave cannot perform I/O. All TLS records are forwarded to the trusted TLS interface, which first performs the TLS handshake using the most recent server certificate. Afterwards, it decrypts/encrypts all incoming/outgoing TLS records. As such, the trusted TLS interface is the endpoint of a secure channel from the user application to the enclave.

Request Handler. The request handler component parses each incoming request, checks the syntax, uses the identity information in the client certificate to allocate the request to a user U, and processes requests as outlined in Algorithm 11. During processing, it uses *internal operations* (see Table 8.4), which are provided by the access control and file manager components described later. We omit error handling in the algorithms for brevity.

Operation	Description
$F \leftarrow \mathsf{toFile}(path)$	Get file F corresponding to path $path$
$path' \leftarrow \mathsf{parent}(path)$	For path $path$, get parent directory's path $path'$
$\mathsf{write}(path, con)$	Create or update file at path $path$ with content con
$con \leftarrow \mathsf{read}(path)$	Read file at path $path$
$\{0,1\} \leftarrow \mathsf{existsF}(path)$	Check if file with path $path$ exists, i.e., $\exists\, F \in \boldsymbol{FS}\colon \mathsf{toFile}(path) = F$
$\{0,1\} \leftarrow \mathsf{existsG}(G)$	Check if group G exists, i.e., $\exists\, G' \in \boldsymbol{G}\colon G == G'$
$\{0,1\} \leftarrow \mathsf{isDir}(path)$	Check if file with path $path$ is a directory i.e., $\exists\, F \in \boldsymbol{F}^D\colon \mathsf{toFile}(path) == F$
$C \leftarrow \mathsf{AE\_Enc}(SK, V)$	Encrypt value V with an AE scheme under key SK
$V \leftarrow \mathsf{AE\_Dec}(SK, C)$	Decrypt the ciphertext C with an AE scheme under key SK
$\{0,1\} \leftarrow \mathsf{authF}(U, P, F)$	Check if user U has permission P on file F, i.e., $\exists\, G\colon (U,G) \in \boldsymbol{R}^G \wedge ((P,G,F) \in \boldsymbol{R}^P \vee (G,F) \in \boldsymbol{R}^{FO})$
$\{0,1\} \leftarrow \mathsf{authG}(U, G)$	Check if user U is allowed to change group G, i.e., $\exists\, G'\colon (U,G') \in \boldsymbol{R}^G \wedge (G',G) \in \boldsymbol{R}^{GO}$
$\mathsf{updateRel}(\boldsymbol{R}, \boldsymbol{R}')$	Update relation $\boldsymbol{R}$ to $\boldsymbol{R}'$

Table 8.4: SeGShare's handling of internal operations.

SeGShare achieves separation of authentication and authorization (see obj. F8) by allocating a request to U based on the identity information in the client certificate and using U for authorization decisions. Furthermore, the combination of operations outlined in Algorithm 11 allows a user to share a file or directory with individual users (using their default groups) and groups (see obj. F1 and P2); dynamically change permissions and group memberships (see

Algorithm 11 SeGShare's external requests.

▷ User U wants to create a directory at path $path$
function putDir($U, path$);
  $path' =$ parent($path$)
  if isDir($path$) and !existsF($path$) and existsF($path'$) **then**
    if $path' ==$ "/" or authF(U, P^w, toFile($path'$)) **then**
      updateRel($\boldsymbol{R}^{FO}, \boldsymbol{R}^{FO} \cup (G^U$, toFile($path$)))
      write($path$,"")
      $con \leftarrow$ AE_Dec($SK^{f\prime}$, read($path'$))
      write($path'$, AE_Enc($SK^{f\prime}, con + path$))

▷ User U wants to create or update a file at path $path$ with content con
function putFile($U, path, con$)
  $path' =$ parent($path$)
  If !isDir($path$) and (($path' ==$ "/") or
    (existsF($path'$) and authF(U, P^w, toFile($path'$))) or
    (existsF($path$) and authF(U, P^w, toFile($path$)))) **then**
    if !existsF($path$) **then**
      $con' \leftarrow$ AE_Dec($SK^{f\prime}$, read($path'$))
      write($path'$, AE_Enc($SK^{f\prime}, con' + path$))
      updateRel($\boldsymbol{R}^{FO}, \boldsymbol{R}^{FO} \cup (G^U$, toFile($path$)))
    write($path$, AE_Enc(SK^f, con))

▷ User U requests file content if toFile($path$) $= F \in \boldsymbol{F}^C$ and directory listing if toFile($path$) $= F \in \boldsymbol{F}^D$
function get($U, path$)
  if authF(U, P^r, toFile($path$)) **then**
    return AE_Dec(SK^f, read($path$))

▷ User U wants to set permission P for group G for file at $path$
function setP($U, path, G, P$)
  if authF(U,"", toFile($path$)) **then**
    updateRel($\boldsymbol{R}^P, \boldsymbol{R}^P \cup (P, G$, toFile($path$)))

▷ User U wants to add user U' to group G
function addU(U, U', G)
  if !existsG(G) **then**
    updateRel($\boldsymbol{G}, \boldsymbol{G} \cup G$)
    updateRel($\boldsymbol{R}^{GO}, \boldsymbol{R}^{GO} \cup (G^U, G)$)
  if authG(U, G) **then**
    updateRel($\boldsymbol{R}^G, \boldsymbol{R}^G \cup (U', G)$)

▷ User U wants to remove user U' from group G
function rmvU(U, U', G)
  if authG(U, G) **then**
    updateRel($\boldsymbol{R}^G, \boldsymbol{R}^G \setminus (U', G)$)

obj. F2 and F3); and set separate read and write permissions (see obj. F4). None of the listed operations requires any interaction with other users (see obj. F6). Updates of file and group ownerships are not listed, but the operations only require a straightforward update of $\boldsymbol{R}^{FO}$ and $\boldsymbol{R}^{GO}$. These updates allow the setting of multiple file and group owners (see obj. F7).

Access Control. The access control component is responsible for relation updates (internal operation updateRel) and access control checks (internal operations authF and authG). For both tasks, it uses the file manager components to read and write the required relations.

File Managers. The trusted and untrusted file manager components handle all files stored in untrusted memory. The trusted file manager component encrypts/decrypts the content of

all files that should be written/read with AE_Enc/AE_Dec using a unique file key SK^f per file. The file key is derived from a root key SK^r, which the trusted file manager generates and seals (see Section 3.1.4) on the first enclave start and unseals on subsequent enclave starts. All encrypted data is passed/received to/from the untrusted file manager component, which handles the actual memory access (internal operations read and write). The file managers handle the following file types:

1. Each $F \in \boldsymbol{FS}$ is stored as a regular file.
2. For each $F \in \boldsymbol{FS}$, an ACL file is stored under F's path appended with a suffix, e.g., ".acl". This ACL file stores F's access permissions ($\boldsymbol{R}^P$) and file ownerships ($\boldsymbol{R}^{FO}$).
3. One group list file stores all present groups ($\boldsymbol{G}$)
4. For each user $U \in \boldsymbol{U}$, a member list file stores U's group memberships ($\boldsymbol{R}^G$) and also keeps track of U's group ownerships ($\boldsymbol{R}^{GO}$).

The first two file types are stored in the so-called *content store*, the latter two in the *group store*. In the OS's file system, the two stores are two independent directories stored on disk. The files in the content store are stored in dictionaries according to the structure given by their paths. A root directory file stores a list of first level children. The files in the group store are stored flat and a root directory file stores a list of all contained files. This separation adds an extra layer of security and improves the performance as file, directory, and permission operations are independent of group operations.

The content of ACL files, member list files, and the group list file are kept sorted. Thus, a permission update only requires one decryption of the corresponding ACL, a logarithmic search, one insert or update operation, and one encryption of the ACL. Membership updates require the same operations on one member list file and (in some cases) on the group list file. Thus, permission and membership revocations do not require re-encryption of content and directory files (see obj. P3), and they are performed immediately (see obj. S4). Each $F \in \boldsymbol{FS}$ is stored in one encrypted file and F is accompanied by one encrypted ACL file. Consequently, the number of ciphertexts is constant for each content and directory file (see obj. P4). Obviously, the same encrypted content or directory file can be accessed by different groups (see obj. P5). The confidentiality and integrity of content files, permissions, existing groups, and group memberships are protected by encrypting the corresponding files with an AE scheme (see obj. S1 and S2). Note that SeGShare is optimized for dynamic groups, but it is inefficient to remove a complete group as the member list of each user has to be checked and possibly modified.

8.4 Extensions

In this section, we present SeGShare extensions that fulfill the remaining objectives: data deduplication; inherited permissions; filename and directory structure hiding; individual file rollback protection; and file system rollback protection. Additionally, we discuss SeGShare replication and file system backups. The combination of extensions is explained if it is not straightforward.

8.4.1 Data Deduplication

The goal of data deduplication is to save storage cost by only storing a single copy of redundant objects, which can either be files [255], [256] or blocks. Blocks can be further divided in fixed-size [257] or variable-size [258] blocks. Deduplication can be done client side, i.e., users ask the server if a file is already present and only upload the whole file if necessary, or server-side, i.e., users always upload and the server performs deduplication.

SeGShare is compatible with all mentioned deduplication alternatives. However, we focus on a mechanism for file- and server-based deduplication, because it does not require additional client-side processing, prevents client-side attacks [259], [260], and has the smallest leakage of the server-side approaches [261].

Data deduplication is enabled in SeGShare (see obj. F9) by introducing a third store, denoted by *deduplication store*, and modifying the trusted file manager. For each uploaded content file, the trusted file manager performs the following steps:

1. Temporarily store the file in the deduplication store under a unique random name.
2. Calculate $t \leftarrow \mathsf{MAC\_Tag}(SK^r, fileCon)$ with $fileCon$ being the file content of the temporarily stored file.
3. Convert t to a hex string `hName`.
4. If no file with the name `hName` is present in the deduplication store, rename the temporary file to `hName`. Otherwise, remove the temporary file.
5. Add a content file to the content store as presented in Section 8.3.2 with the slight modification that the content file is not filled with the actual file content but with `hName` (comparable to symbolic links in file systems).

For each request to a content file, the trusted file manager accesses the file in the content store and follows the indirection to the file in the deduplication store.

Our server-side deduplication is different from any scheme presented in related work: plaintext data is deduplicated and only a single copy is encrypted. This is possible because the enclave has access to the file keys. Furthermore, the scheme supports deduplication of data belonging to different groups and immediate membership revocation without re-encryption.

8.4.2 Inherited Permissions

Permissions for any file $F \in \boldsymbol{FS}$ can be inherited from a parent directory (see obj. F10) with the following extension of SeGShare. The user application and request handlers are extended with a new request to add/remove F to/from the inherit relation ($\boldsymbol{R}^I$). The access control component only allows a file owner to execute such requests, and the trusted file manager adds/removes an *inherit flag* to/from F's ACL file. If the inherit flag is not set in F's ACL file, access control checks for F are performed by authF as defined in Table 8.4. Otherwise, a permission P defined for a group G on F has precedence over a permission P' defined for G on F's parent F'. In other words, if F' is F's parent and the inherit flag is set, authF uses the following predicate: $\exists G\colon (U,G) \in \boldsymbol{R}^G \wedge ((P,G,F) \in \boldsymbol{R}^P \vee (G,F) \in \boldsymbol{R}^{FO} \vee ((P,G,F) \notin \boldsymbol{R}^P \wedge (P,G,F') \in \boldsymbol{R}^P))$[1].

8.4.3 Filename and Directory Structure Hiding

To protect the confidentiality of the file system structure (see obj. S1), this extension hides all filenames and the directory structure. A change of the trusted file manager is sufficient: before passing a path $path$ to the untrusted file manager, calculate $t \leftarrow \mathsf{MAC\_Tag}(SK^r, path)$ and convert t to its hexadecimal representation. As a result, all files are stored in a flat directory structure at a pseudorandom location. Note that SeGShare stores the original path in the directory files. Therefore, directory listing is still possible.

As performance optimization, the extension can use a fixed number of x characters of t's hexadecimal representation as a directory name and the remaining characters as filename. As a result, all files are spread across 16^x subdirectories. A similar technique is used in the version control system Git [262].

1 The difference to authF from Table 8.4 is highlighted in green.

8.4.4 Rollback Protection for Individual Files

The trusted file manager encrypts the content of all files with an AE scheme, and thus guarantees confidentiality and integrity on each file individually. However, an attacker can perform a *rollback attack* on each encrypted file, i.e., the attacker uses an outdated version of an encrypted file to replace the current version. Not preventing such a rollback can have severe consequences, e.g., an old member list could enable a user to regain access to files for which the permissions were previously revoked.

To protect the rollback for individual files (see obj. S5), the extension that we describe in this section uses a Merkle tree variant (see Section 2.2.8). For brevity, we only describe how to protect files in the content store, but protecting the group store (and deduplication store) is a straightforward adaptation.

In our Merkle tree variant, each content file, ACL file, and empty directory file is represented by a leaf node, and each non-empty directory file is represented by an inner node. Each leaf node stores a hash that is a combination of hashes over the file path and the file content. Each inner node stores a hash that is a combination of the hash of all children (content files, ACL files, and directory files), a hash over the directory path, and a hash over the directory content (children list). The path is part of the combined hashes for three reasons: (1) files with the same content but stored at different paths should have different hashes, (2) the hash of a file should change if a file is moved, and (3) an empty directory should have a hash.

Before encryption, the trusted file manager prepends the content of content files, ACL files, and directory files with the combined hashes. After decryption, the trusted file manager reads the hashes from there. As each file stores its own hash, we denote each file representing a leaf node, inner node, sibling node, and child node of the tree by *leaf file*, *inner file*, *sibling file*, and *child file*, respectively. The tree's root hash is stored in the root directory file, which we denote by *root file* in this section. Figure 8.2 shows a file system example and the corresponding hash tree.

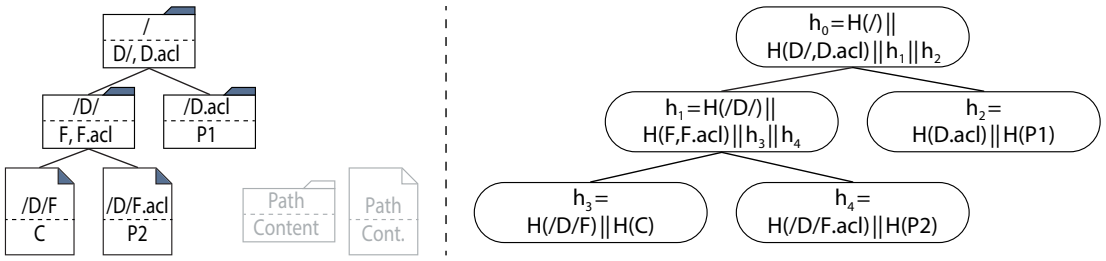

Figure 8.2: Example of a file system (left) and its corresponding hash tree (right). || denotes the concatenation of hashes.

With a regular Merkle tree, a leaf file update or a leaf file addition leads to an update of all hashes on the path from the leaf file to the root file. In each level of the tree, all sibling files have to be accessed to combine their hashes. On a leaf file read, a validation is performed starting from the leaf file to the root file, which also accesses all sibling files. We propose two modifications to optimize this process:

1. Instead of a cryptographic hash function H, we use a set hash function SH for all individual hashes, SH_NI_Add to combine hashes, and SH_SS_Sub to subtract hashes (see Section 2.2.7).

   This modification improves leaf file updates and additions, because it allows hash updates of each inner file by using SH_SS_Sub to subtract the hash of the no longer valid child file and SH_NI_Add to add the hash of the new child file, without accessing any sibling file.

2. Depending on the number of child files, each inner file stores one *main hash* and multiple *bucket hashes*. The main hash stores a combination of the hashed file path, the hashed file content, and the file's bucket hashes. Each bucket hash stores a combination of the main hashes of specific child files. The mapping of child files to bucket hashes is done via a hash calculated over the child files' path.

   This change slightly deteriorates update performance as two hashes have to be updated for every level of the tree. However, for leaf file validation, it is sufficient to recalculate and compare a single bucket hash per tree level (using SH_Comp), which only requires an access to all files in the same bucket.

Note that the individual file rollback protection extension does not protect a rollback of the root file, which we consider a *rollback of the whole file system* for which a mitigation is presented in the next section. However, the extension protects the integrity of the file system structure (see obj. S2).

8.4.5 Rollback Protection for Whole File System

Even with the protection from Section 8.4.4, an attacker can still roll back the whole file system. The key to mitigate this rollback is to protect the root hash against rollbacks as it represents a state of the complete file system. Based on TEE functionality, we propose two solutions to protect the root hash and with it prevent rollbacks of the whole file system (see obj. S5).

First, if the TEE offers a protected memory that can only be accessed by a specific enclave and is persisted across restarts, it is sufficient to write/read the root hash into/from this memory, instead of storing it in the root file. Second, if the TEE offers a monotonic counter that can only be accessed by a specific enclave and is persisted across restarts, we propose the following. On each file update, the trusted file manager increments the TEE's monotonic counter and writes the new counter value into the root file before encryption. On validity checks of the root hash, it compares the TEE's monotonic counter with the counter value stored in the root file.

The group store's (and deduplication store's) root hash have to be protected by the same mechanism to protect the rollback of all permissions (and deduplicated files).

Note that Intel SGX provides monotonic counters, but the current implementation has issues [263]: read and write operations on the non-volatile memory are slow; the counter wears out after approximately 1 million increments; and the non-volatile memory resides outside the processor package making it vulnerable to bus tapping and flash mirroring attacks. Until a better hardware-based monotonic counter is available, one can use ROTE [263]. LCM [264] is another alternative, but it requires periodic interactions with the majority of users.

8.4.6 SeGShare Replication

As we show in our evaluation section, SeGShare has a very low latency. Nonetheless, it might be necessary to deploy SeGShare on multiple servers if many users want to use the file sharing service. Assuming that all enclaves access the data from one central data repository, two changes are necessary for SeGShare replication: (1) the untrusted file manager must be extended to access data from the central data repository, and (2) all enclaves need access to the same root key SK^r.

The first change is only an implementation issue and therefore we only discuss the second in detail. We denote enclaves that already have SK^r by *root enclaves* and the others by *non-root enclaves*. We propose that the CA tasks one enclave with the generation of SK^r during the provisioning of the server certificate. The CA provides all other enclaves with addresses of root enclaves during server certificate provisioning. Each non-root enclave randomly selects one

root enclave and performs remote attestation with it. If the measurements of both enclaves are equal, the non-root enclave is assured to communicate with another enclave that was compiled for the same CA, as the CA's public key is hard-coded. During remote attestation, a secure channel is established between the enclaves, and the root enclave transfers SK^r to the non-root enclave using this channel.

SeGShare replication is also useful for file system owners, which might be afraid to lose access to their files, because SK^r is only accessible by a single enclave. With the proposed method, SK^r is contained inside trusted enclaves at all time, but still usable on multiple replicas.

To combine the whole file system rollback protection and SeGShare replication, it is necessary to use a non-local protected memory or monotonic counter for each store. We note that locking problems and data storage replication is beyond the scope of this dissertation.

8.4.7 File System Backup

SeGShare supports file system backups in a straightforward manner: the cloud provider only has to copy the files on disk. Backup restoration depends on the enclave that handles the restored data. If the enclave is the same that wrote the files in the first place, it poses the required decryption key. Otherwise, the SeGShare replication process described in Section 8.4.6 is necessary.

Restoration becomes more complicated if the whole file system rollback protection is active, because it might be necessary to restore an old state. We propose that the CA can send a signed reset message to the enclave for this case. The enclave checks the validity of the message's signature, reads the stored hashes from the root files of the various stores, recalculates the root hashes, and compares the hashes. Assuming a successful check and the monotonic counter–based rollback solution, the enclave overwrites the stored monotonic counter with the TEE's current monotonic counter.

8.5 Implementation

Our prototype is implemented in C/C++ using the Intel SGX SDK in version 2.5. The prototype follows the WebDAV standard [265], which is an extension to the HTTP standard designed for change and permission management of web resources. WebDAV makes the prototype compatible with existing clients on Android [266], iOS [267], Windows [268], Mac [269], and Linux [270]. The secure channel to transfer messages is established with TLSv1.2 using the ECDHE-RSA-AES256-GCM-SHA384 cipher suite, and the Set-XOR-Hash construction is used as set hash function (see Section 2.2.7). All requests are parsed by an imported HTTP parser [271]. For all file accesses, we use Intel's Protected File System Library (see Section 3.1.6). The prototype contains the following extensions: filename and directory structure hiding, and rollback protection for individual files.

Our prototype tackles three performance problems:

1. A main performance bottleneck is the TLS stack. Unfortunately, publicly available Intel SGX–enabled TLS stacks [272], [273] are mainly designed for embedded scenarios and do not provide the desired performance. Intel only provides an Intel SGX optimized version of OpenSSL's cryptographic library, without networking capabilities. Our prototype combines Intel's cryptographic library with the network part of OpenSSL [274] in version 1.1.1c.
2. Switches into and out of the enclave have a high overhead [67]. To mitigate this problem, our prototype uses switchless calls (see Section 3.1.7) for our TLS library and for Intel's Protected File System Library.

3. Intel SGX has restricted EPC memory. Our prototype addresses this problem via streaming, i.e., users send and receive small, fixed-size chunks and the enclave processes one chunk at a time. The same chunk-wise processing is done for all storage operations. Thus, the enclave only requires a small, constant size buffer for each request.

8.6 Evaluation

Based on the SeGShare design presented in Section 8.3, the SeGShare extensions described in Section 8.4, and the implementation details explained in Section 8.5, we provide a security, storage, and performance evaluation for SeGShare in this section.

8.6.1 Security Evaluation

For our security evaluation, we focus on SeGShare's main security objective: end-to-end protection of user files (see obj. S3). The basis for this objective is SeGShare's setup phase and mutual authentication during runtime (see Section 8.3.1): the trusted CA securely provisions certificates only to valid SeGShare enclaves and user applications, SeGShare enclaves only accept user applications, which present a valid client certificate, and user applications only send files to an enclave, which present a valid server certificate. Based on this trust, a secure TLS channel is established between user applications and SeGShare enclaves protecting all messages in transit. The enclave has plaintext access to messages and file contents, but the enclave protects the processing and processed data. The enclave also enforces authorizations according to our access control model (see Table 8.1), which enforces that an attacker is restricted to the union of permissions of the users under her control. As discussed in the design and extensions sections, the enclave protects the integrity and confidentiality of all files stored in untrusted storage (see obj. S1 and S2), enforces revocations immediately (see obj. S4), and mitigates rollbacks (see obj. S5). Overall, user files are protected in transit, during processing, and during storage.

We note that SeGShare's security hinges on a trusted enclave, which we assume in our attacker model. Nevertheless, we kept the enclave code as small as possible, as this reduces the probability of security-relevant implementation errors, unintended leakages, hidden malware, and side-channel leakages. Besides the Intel SGX SDK, the enclave has only 8441 LOC. 2099 of these are due to the imported HTTP parser and 2376 due to our TLS implementation, which can be replaced by a formally verified implementation [275].

8.6.2 Storage Evaluation

The storage overhead for each file $F \in \boldsymbol{FS}$ depends on F's ACL file and the overhead introduced by Intel's Protected File System Library (see Section 3.1.6) for both files. Remember that the size of an ACL file depends on the number of file owners and group permissions, and Intel's library uses a Merkle tree with 4 kB nodes. Our prototype uses 32 bit for the number of file owners and the inheritance flag, and 32 bit for each file owner and group permission. A 10 MB plaintext file together with its ACL file requires 10.11 MB and 10.15 MB encrypted storage, if the ACL contains up to 95 and 1119 entries, respectively. This corresponds to a relative storage overhead of 1.12% and 1.48%. A 200 MB plaintext file with the same ACL files requires 202.09 MB and 202.13 MB encrypted storage corresponding to 1.05% and 1.06% overhead.

8.6.3 Performance Evaluation

For the performance evaluation, we measure SeGShare's latency for file uploads and downloads, membership additions and revocations, and permission additions and revocations. Additionally, we measure the latency introduced by the rollback protection for individual files. We perform all latency measurements with two machines hosted at Microsoft Azure [172]:

- A client with 2 vCPU cores of an Intel Xeon CPU E5-2673 v4 @ 2.30 GHz and 8 GB RAM located in the Central US region.
- A server with 4 Intel SGX–enabled vCPUs of an Intel Xeon E-2176G @ 3.70 GHz and 16 GB RAM located in the East US region.

We measure the latency from the start of the request at the user application until the complete response arrived. Consequently, the reported results also include network latency. SeGShare interleaves sending and processing due to our streaming technique; hence, we do not present the pure network latency of each request. We present all latencies as mean of 100 runs with a 95% confidence interval (in our plots).

In the first experiment, we upload and download files with sizes from 1 MB to 200 MB to SeGShare. For the baseline, we execute the same test with two TLS-enabled—but plaintext storing—WebDAV servers: Apache HTTP Server [276] in version 2.4 and nginx [277] in version 1.17.8. Figure 8.3 shows that uploads and downloads of a 200 MB file, on average, take 2.39 s and 2.17 s for SeGShare, 4.74 s and 2.62 s for the Apache server, and 1.84 s and 0.93 s for the nginx server. Overall, the upload and download performance of SeGShare is in between the two plaintext WebDAV servers.

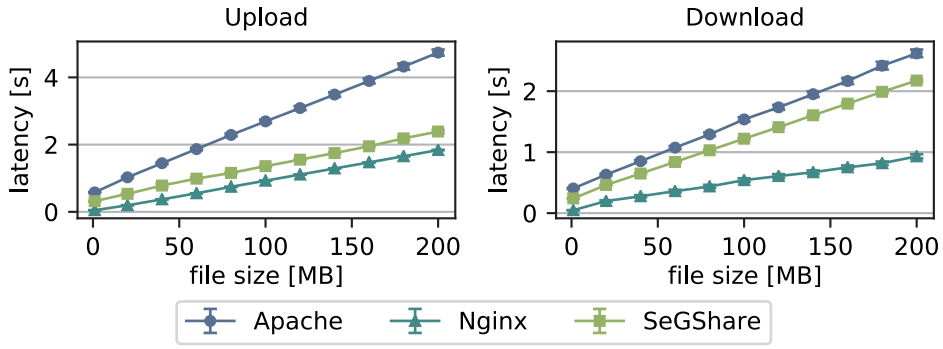

Figure 8.3: Mean latency of 1000 up-/downloads with different file sizes.

In the second experiment, we measure the latency to add/revoke a user to/from his first group. These membership operations only affect the member list file of the user. Therefore, they are independent of the number of permissions $|\boldsymbol{R}^{P}|$, stored files $|\boldsymbol{FS}|$, inherit flags $|\boldsymbol{R}^{I}|$, file owners $|\boldsymbol{R}^{FO}|$, group owners $|\boldsymbol{R}^{GO}|$, and the file sizes. Furthermore, these operations are also independent of the number of members the group had before, because SeGShare does not explicitly store a list of all group members. On average, it takes 154.05 ms and 153.40 ms for additions and revocations, respectively.

In the third experiment, we measure the latency of adding/revoking a user to/from a group if the user is already a member of several groups. Again, only the member list file is affected, and the latency is independent of $|\boldsymbol{R}^{P}|, |\boldsymbol{FS}|, |\boldsymbol{R}^{I}|, |\boldsymbol{R}^{FO}|, |\boldsymbol{R}^{GO}|$, and the file sizes. However, the latency now depends logarithmically on the number of the user's group memberships, because a logarithmic search is necessary to insert or remove a group membership into or from the member list file. As presented in Figure 8.4, this dependency is negligible in the mean latency even up to 1000 group memberships: the latency is between 150.29 ms and 150.92 ms for additions, and between 150.11 ms and 151.13 ms for revocations.

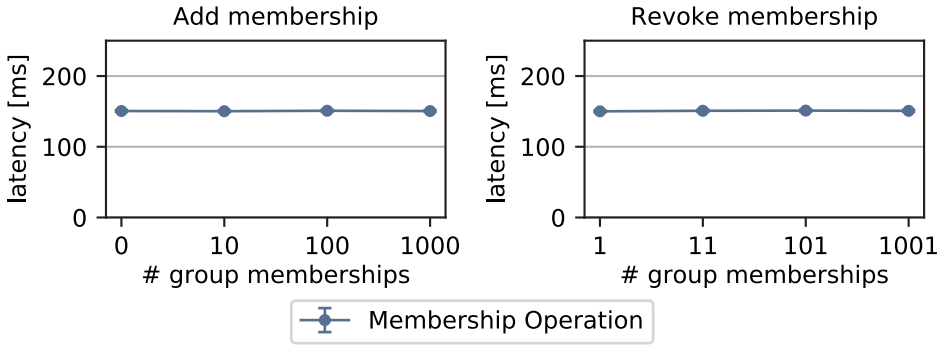

Figure 8.4: Mean latency of adding/revoking a user to/from a group (x-axis: number of memberships before operation).

In the fourth experiment, we measure the latency of adding/revoking a group permission to/from a file if several groups already have access. For these operations, only permissions in the ACL file are accessed; thus, the latency is independent of the number of group memberships $|\boldsymbol{R}^G|, |\boldsymbol{FS}|, |\boldsymbol{R}^I|, |\boldsymbol{R}^{FO}|, |\boldsymbol{R}^{GO}|$, and the file sizes. The latency depends logarithmically on the number of groups having access, but Figure 8.5 shows that this dependency is again negligible in the total latency.

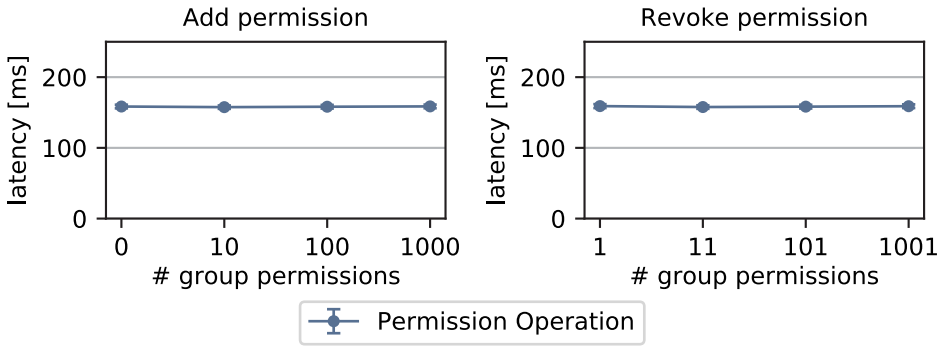

Figure 8.5: Mean latency of adding/revoking a group permission to/from a file (x-axis: number of permissions before operation).

In the last latency experiment, we evaluate the overhead of SeGShare's individual file rollback protection extension with the following measurement for $x \in [0, 14]$. As preparation, we upload $(2^x - 1)$ times a 10 kB file to SeGShare according to two directory structures: (1) directories are organized as a binary tree and each leaf contains one file and (2) all files are stored flat under the root. Then, we measure the upload and download of one additional 10 kB file. Figure 8.6 shows that due to our optimizations, the overhead introduced for uploads is negligible in the total latency. The minimal, average download latencies for directory structure (1) and (2) are 111.65 ms and 111.65 ms. Even for 16,384 files, the average latency only increases to 115.93 ms and 121.95 ms.

8.7 Summary

In this chapter, we showed how a TEE-protected file system can be used to build a secure, efficient, outsourced system for group file sharing supporting large and dynamic groups—SeGShare. This system protects the confidentiality and integrity of content files, the file system structure, permissions, existing groups, and group memberships. Among other features, it

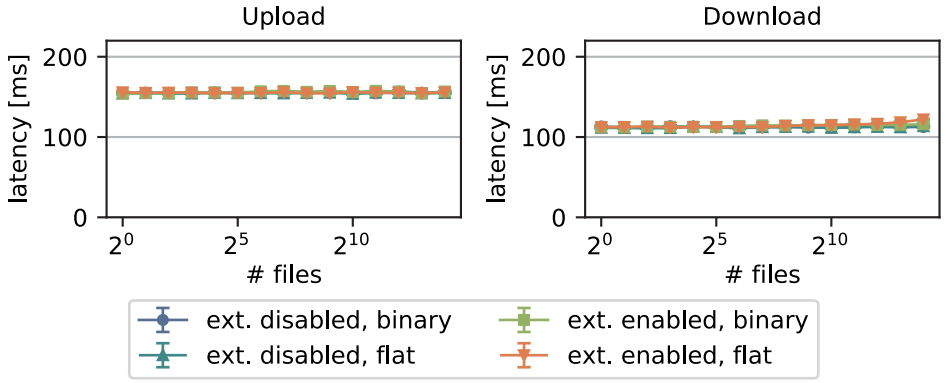

Figure 8.6: Mean latency with enabled and disabled individual file rollback protection extension and different directory structures.

enforces immediate permission and membership revocations; supports deduplication; mitigates rollback attacks; and provides separation of authentication and authorization. SeGShare's enclave code comprises 8441 LOC minimizing common pitfalls in deploying code to TEEs. A 200 MB plaintext file requires only 202.13 MB encrypted storage, even if it is shared with more than 1000 groups. Due to our optimized TLS stack and chunk-wise streaming, it takes under 2.4 s to upload and download a 200 MB plaintext file, which is faster than Apache WebDAV serving plaintext files in the same setup. Permissions and memberships updates require under 170 ms, independent of the number of stored files, file sizes, number of group members, number of user permissions, and groups sharing a file.

9 Conclusion

In the following, we first summarize the chapters of this dissertation. Then, we conclude the dissertation with an outlook on potential future research avenues, in Section 9.1.

In our introduction (Chapter 1), we established that security is one of the main challenges for companies that want to outsource data processing to the cloud. We listed approaches trying to solve this security challenge purely based on cryptography. As these approaches have severe shortcomings, we proposed that the untrusted cloud provider uses a TEE for secure and efficient data processing. In particular, the TEE should be used to process data structures, and the resulting TEE-protected data structures can be embedded in outsourced systems. Based on this proposition, we formulated our research question, explored various aspects of the research question, and listed the three systems on which we want to test our proposition.

In Chapter 2, we introduced the notation that we use consistently throughout this dissertation. Additionally, we formally introduced cryptographic primitives that are required for the remainder of the dissertation, e.g., authenticated encryptions, message authentication codes, and set hash functions.

As TEEs are an essential aspect of this dissertation, we dedicated Chapter 3 to their description. After some important definitions, we listed seven capabilities that we expect from a TEE. In principle, the systems we present in this dissertation can be implemented with any TEE supporting these seven capabilities. However, during the timeframe in which the research for this dissertation was done, only Intel SGX fulfilled all capabilities. Therefore, we elaborated how Intel SGX fulfills the capabilities, described its TCB, and highlighted its auxiliary capabilities. As many attacks on Intel SGX were presented in the last years, we explained these attacks and discussed mitigations against them. In particular, we concluded that a small enclave size is key to mitigate most attacks. We concluded Chapter 3 with a differentiation to technologies related to TEEs and a classification of commercially available TEEs according to our list of seven capabilities.

At this point, we had the necessary foundations for an in-detail exploration of related approaches enabling secure, outsourced data processing. In Chapter 4, we explained related approaches, reviewed whether these approaches are compatible with our secure, outsourced, TEE-based data processing scenario, and checked if they provide strong security, high efficiency, and arbitrary processing capabilities. We demonstrated that the approaches based purely on cryptography cover other scenarios and/or they fail in at least one of these aspects. We also presented TEE-based approaches that put entire applications or individual, stateless operations into an enclave. A benefit of these approaches is that they provide a generic solution, which can be applied to legacy applications. However, protecting entire applications leads to a large TCB, makes it difficult to capture the exact leakage, and impedes attack mitigation strategies. Protecting individual, stateless operations does not suffer from these problems, but leaks the result of each operation.

In Chapter 5, we used the result of the related work section to introduce our approach combining strong security, high efficiency, and arbitrary processing capabilities—the TEE-protected processing of data structures. We differentiated this approach from the individual, stateless operations approach, and we listed five design principles that we want to adhere

to for our TEE-protected data structures. In Chapter 5, we also explored our security and performance assessment methodology, which we used in the following sections to validate our research question.

Chapter 6 is the first main chapter of this dissertation. In it, we explained how TEE-protected B^+-trees were used to design HardIDX—a secure, efficient, outsourced system for index searches. We presented two HardIDX constructions with different memory-management strategies: The first construction loads the whole B^+-tree in the enclave memory and the enclave processes search queries thereafter. The second construction initially loads only the root node into the enclave memory and the enclave loads required nodes on demand during the tree traversal. For both constructions, we provided a formal security proof. Additionally, we presented latency measurements for both constructions, and showed that the second construction requires about 1 ms to perform range queries on B^+-trees containing 50 million encrypted entries.

In Chapter 7, we explored a second TEE-protected data structure: database dictionaries. Using this data structure, we designed EncDBDB—a secure, efficient, outsourced, dictionary-encoding–based, column-oriented, in-memory database supporting analytic queries on large datasets. EncDBDB provides nine encrypted dictionary types with distinct security, performance, and storage efficiency trade-offs. In our security evaluation, we classified the security of each encrypted dictionary type based on known security schemes or definitions. In our performance evaluation, we presented latency measurements for each encrypted dictionary type. On average, EncDBDB executes range queries over columns with almost 11 million encrypted rows in less than 13 ms, even without frequency leakage and with bounded order leakage.

We presented our third TEE-protected data structure, a file system, in Chapter 8. Our third system, SeGShare uses a TEE-protected file system to provide secure, efficient, outsourced group file sharing supporting large and dynamic groups. Among other features, SeGShare also enforces immediate permission and membership revocations; supports deduplication; mitigates rollback attacks; and provides separation of authentication and authorization. We classified the security of SeGShare using five security objectives and showed that no related approach can fulfill these objectives. Additionally, we established that the same is true for five important performance objectives, and we presented latency measurements for important group file sharing operations. Due to our optimized TLS stack and chunk-wise streaming, it takes under 2.4 s to upload and download a 200 MB plaintext file, which is faster than Apache WebDAV serving plaintext files in the same setup.

In conclusion, Chapter 6, 7, and 8 have shown that the integration of TEE-protected data structures in outsourced systems is a valid approach. In all cases, we answered our research question by providing lower bounds of security and corresponding upper bounds on performance. Moreover, we demonstrated that our approaches provide a favorable trade-off regarding security and efficiency compared to related approaches.

9.1 Outlook

The results of this dissertation open a number of interesting avenues for future work. In the following, we briefly explore three avenues: extension of our systems, adoption of our approach to new TEEs, and protection of further data structures.

The three systems presented in our main chapters can be extended in various directions:

- HardIDX uses B^+-trees for range searches and thus trivially supports equality searches. With minor modifications, HardIDX could also support prefix, suffix, or substring searches. Additionally, HardIDX could be embedded in a larger data analysis framework for secure and efficient queries. In this case, HardIDX should also support secure and efficient insertions, updates, and deletions.

- EncDBDB's current implementation support equality and range queries, but the implementation could be easily extended to support further described functionality, i.e., joins, insertions, deletions, updates, counts, aggregations, and average calculations. Furthermore, integrity protection for whole columns instead of individual values would be a valuable extension to EncDBDB.
- The current version of SeGShare can be shipped as a standalone application for secure group data sharing. With slight modifications, SeGShare could also hide the identity of the users from the cloud provider and from other users accessing the data. Such a system is valuable in various settings, e.g., in a military organization, the files *and* the users with a specific data access are confidential; in a business acquisition setting, the corporate files *and* the bidding parties might be confidential; and in a hospital, a patient's files *and* the doctors with access might be confidential, because the doctors' specialty can leak sensitive information.

Quite recently, AMD SEV-SNP was unveiled, and it is the only TEE other than Intel SGX which supports the capabilities required by this dissertation. In contrast to Intel SGX, the RAM region protected by AMD SEV-SNP is only limited by the size of the RAM. Most certainly, comparable TEEs will be presented by multiple vendors in the near future. With such TEEs, it might be useful to adapt some of our designs. For adaptations to large protected RAM regions, we claim that only the data size processed by an enclave should be increased. The enclave size, i.e., the size of an enclave's source code in LOC, should be kept small. This is important, because a large enclave size leads to a large TCB, makes it difficult to capture the exact leakage, and impedes attack mitigations. A possible adaptation of HardIDX is the following: the enclave caches the top-level nodes of the B^+-tree, loads more and larger nodes, and parallel-processes multiple nodes. This would strengthen HardIDX's security and improve its performance.

In this dissertation, we cover three TEE-protected outsourced data structures. Besides these three, our approach could be used for the secure and efficient processing of other data structures. For instance, TEE-protected hash tables would provide a very efficient alternative for key-to-value(s) mappings; TEE-protected suffix trees would provide efficient string operations; and TEE-protected graphs could be used for various commonly used operations (e.g., shortest path calculations, breadth-first searches, minimum spanning tree calculations).

Bibliography

[1] A. Regalado. (Oct. 2011). Who Coined 'Cloud Computing'?, [Online]. Available: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/ (visited on 09/04/2020) (cit. on p. 3).

[2] IDG. (2020). IDG Cloud Computing Survey, [Online]. Available: https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/ (visited on 09/04/2020) (cit. on p. 3).

[3] Ponemon Institute. (2019). Protecting Data In The Cloud 2019 Thales Cloud Security Study, [Online]. Available: https://cpl.thalesgroup.com/cloud-security-research (visited on 09/04/2020) (cit. on p. 3).

[4] Gartner. (2017). Cloud Strategy Leadership, [Online]. Available: https://www.gartner.com/en/publications/cloud-leadership (visited on 09/04/2020) (cit. on p. 3).

[5] D. R. Stoller. (Feb. 2020). Retail Customer Data Exposure Spotlights Cloud Security Risk (1), [Online]. Available: https://news.bloomberglaw.com/privacy-and-data-security/retail-customer-data-exposure-spotlights-cloud-security-risk (visited on 09/04/2020) (cit. on p. 3).

[6] Security magazine. (Mar. 2020). Data Breach Report: Cloud Storage Exposes 270,000 Users' Private Information, [Online]. Available: https://www.securitymagazine.com/articles/91985-data-breach-report-cloud-storage-exposes-users-private-information?v=preview (visited on 09/04/2020) (cit. on p. 3).

[7] vpnMentor. (2020). Report: Popular Digital Wallet Exposes Millions to Risk in Huge Data Leak, [Online]. Available: https://www.vpnmentor.com/blog/report-keyring-leak/ (visited on 09/04/2020) (cit. on p. 3).

[8] vpnMentor. (2020). Report: Exercise App Exposes Private User Data in Massive Data Leak, [Online]. Available: https://www.vpnmentor.com/blog/report-kinomap-leak/ (visited on 09/04/2020) (cit. on p. 3).

[9] IDC. (2020). IDC Cloud Security Survey Highlights, [Online]. Available: https://l.ermetic.com/wp-idc-survey-results (visited on 09/04/2020) (cit. on p. 3).

[10] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and Efficiently Searchable Encryption", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2007 (cit. on pp. 4, 36, 49, 73, 80, 91).

[11] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order Preserving Encryption for Numeric Data", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2004 (cit. on pp. 4, 36, 73, 80).

[12] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-Preserving Symmetric Encryption", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2009 (cit. on pp. 4, 36, 37, 49, 73, 80).

[13] M. Naveed, S. Kamara, and C. V. Wright, "Inference Attacks on Property-Preserving Encrypted Databases", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2015 (cit. on pp. 4, 36, 37, 49, 54, 80, 91).

[14] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart, "Leakage-Abuse Attacks against Order-Revealing Encryption", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2017 (cit. on pp. 4, 37, 54, 80, 91).

[15] M.-S. Lacharité, B. Minaud, and K. G. Paterson, "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2018 (cit. on pp. 4, 38, 54, 80).

[16] R. L. Rivest, L. Adleman, M. L. Dertouzos, *et al.*, "On data banks and privacy homomorphisms", 1978 (cit. on pp. 4, 40).

[17] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices", in *Proceedings of the ACM Symposium on Theory of Computing*, ser. STOC, 2009 (cit. on pp. 4, 40, 73).

[18] C. Gentry and S. Halevi, "Implementing Gentry's Fully-Homomorphic Encryption Scheme", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2011 (cit. on pp. 4, 40).

[19] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic Evaluation of the AES Circuit", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2012 (cit. on pp. 4, 49, 73).

[20] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing", in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP, 2013 (cit. on pp. 5, 20).

[21] V. Costan and S. Devadas, "Intel SGX Explained", IACR Cryptology ePrint Archive, Report 2016/086, 2016 (cit. on pp. 5, 20).

[22] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions", in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP, 2013 (cit. on pp. 5, 20).

[23] Intel. (2014). Intel® Software Guard Extensions Programming Reference, [Online]. Available: `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf` (visited on 09/04/2020) (cit. on pp. 5, 19, 20).

[24] Intel. (2020). Intel® Software Guard Extensions (Intel® SGX) SDK for Linux OS—Developer Reference, [Online]. Available: `https://download.01.org/intel-sgx/sgx-linux/2.11/docs/Intel_SGX_Developer_Reference_Linux_2.11_Open_Source.pdf` (visited on 09/04/2020) (cit. on pp. 5, 20, 24, 42).

[25] Intel. (2020). Intel® Software Guard Extensions (Intel® SGX)—Developer Guide, [Online]. Available: `https://download.01.org/intel-sgx/sgx-linux/2.11/docs/Intel_SGX_Developer_Guide.pdf` (visited on 09/04/2020) (cit. on pp. 5, 20, 24).

[26] Intel Corporation. (2015). Intel® Software Guard Extensions (Intel® SGX), [Online]. Available: `https://software.intel.com/sites/default/files/332680-002.pdf` (visited on 09/04/2020) (cit. on pp. 5, 20).

[27] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution", in *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP, 2013 (cit. on pp. 5, 20).

[28] S. Johnson, V. R. Scarlata, C. V. Rozas, E. Brickell, and F. McKeen. (2016). Intel SGX: EPID Provisioning and Attestation Services, [Online]. Available: `https://software.intel.com/content/www/us/en/develop/download/intel-sgx-intel-epid-provisioning-and-attestation-services.html` (visited on 09/04/2020) (cit. on pp. 5, 20).

[29] D. Vinayagamurthy, A. Gribov, and S. Gorbunov, "StealthDB: a Scalable Encrypted Database with Full SQL Query Support", *Proceedings on Privacy Enhancing Technologies*, PoPETS, 2019 (cit. on pp. 5, 43, 73, 79, 80).

[30] S. Eskandarian and M. Zaharia, "ObliDB: Oblivious Query Processing using Hardware Enclaves", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2019 (cit. on pp. 5, 73, 79, 80).

[31] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset", *Proceedings on Privacy Enhancing Technologies*, PoPETS, 2019 (cit. on p. 5).

[32] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2015 (cit. on pp. 5, 25, 53).

[33] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX", in *Proceedings of the European Workshop on Systems Security*, ser. EuroSec, 2017 (cit. on pp. 5, 25, 53).

[34] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2017 (cit. on pp. 5, 25).

[35] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in Darkness: Return-oriented Programming against Secure Enclaves", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2017 (cit. on pp. 5, 26).

[36] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves", in *Proceedings of the European Symposium on Research in Computer Security*, ser. ESORICS, 2016 (cit. on pp. 5, 26).

[37] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks", in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA, 2017 (cit. on pp. 5, 27).

[38] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX", in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA, 2019 (cit. on pp. 5, 27).

[39] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution", in *Proceedings of the IEEE European Symposium on Security and Privacy*, ser. EuroS&P, 2019 (cit. on pp. 5, 28).

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2018 (cit. on pp. 5, 28).

[41] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling", in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2019 (cit. on pp. 5, 28).

[42] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "HardIDX: Practical and Secure Index with SGX", in *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*, ser. DBSec, 2017 (cit. on pp. 8, 69, 73).

[43] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A. Sadeghi, "HardIDX: Practical and secure index with SGX in a malicious environment", *Journal of Computer Security*, JCS, 2018 (cit. on p. 8).

[44] B. Fuhry, J. Jayanth H A, and F. Kerschbaum, "EncDBDB: Searchable Encrypted, Fast, Compressed, In-Memory Database using Enclaves", arXiv.org, arXiv:2002.05097, 2020 (cit. on p. 8).

[45] B. Fuhry, L. Hirschhoff, S. Koesnadi, and F. Kerschbaum, "SeGShare: Secure Group File Sharing in the Cloud using Enclaves", in *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2020 (cit. on p. 8).

[46] A. Fischer, B. Fuhry, F. Kerschbaum, and E. Bodden, "Computation on Encrypted Data using Dataflow Authentication", *Proceedings on Privacy Enhancing Technologies*, PoPETS, 2020 (cit. on pp. 8, 40).

[47] A. Fischer, B. Fuhry, J. Kussmaul, J. Janneck, F. Kerschbaum, and E. Bodden, "Improved Computation on Encrypted Data using Dataflow Authentication", (under submission), 2020 (cit. on p. 8).

[48] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2rd. CRC press, 2015 (cit. on pp. 10, 13).

[49] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", National Institute of Standards & Technology, 2007 (cit. on pp. 13, 64).

[50] R. L. Rivest, "The MD5 Message-Digest Algorithm", RFC Editor, Tech. Rep. RFC 1321, Apr. 1992 (cit. on p. 15).

[51] "Secure Hash Standard (SHS)", National Institute of Standards and Technology, Tech. Rep. FIPS PUB 180-4, Aug. 2015 (cit. on p. 15).

[52] "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology, Tech. Rep. FIPS PUB 202, Aug. 2015 (cit. on p. 15).

[53] D. Clarke, S. Devadas, M. Van Dijk, B. Gassend, and G. E. Suh, "Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking", in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT, 2003 (cit. on pp. 15, 16).

[54] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest We Remember: Cold Boot Attacks on Encryption Keys", *Communications of the ACM*, 2009 (cit. on p. 19).

[55] OMTP Limited. (2009). Advanced Trusted Environment: OMTP TR1, [Online]. Available: `https://www.gsma.com/newsroom/wp-content/uploads/2012/03/omtpadvancedtrustedenvironmentomtptr1v11.pdf` (visited on 09/04/2020) (cit. on p. 19).

[56] GlobalPlatform Technology. (2018). TEE System Architecture Version 1.2, [Online]. Available: `https://globalplatform.org/specs-library/tee-system-architecture-v1-2/` (visited on 09/04/2020) (cit. on p. 19).

[57] GlobalPlatform Technology. (2011). TEE System Architecture Version 1.0 (cit. on p. 19).

[58] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not", in *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, ser. TrustCom, 2015 (cit. on p. 19).

[59] N. Asokan, J.-E. Ekberg, K. Kostiainen, A. Rajan, C. Rozas, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "Mobile Trusted Computing", *Proceedings of the IEEE*, 2014 (cit. on p. 19).

[60] D. Kaplan, J. Powell, and T. Woller. (2016). AMD MEMORY ENCRYPTION, [Online]. Available: `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf` (visited on 09/04/2020) (cit. on pp. 19, 30, 31).

[61] ARM Limited. (2009). Building a Secure System using Trust-Zone Technology, [Online]. Available: `https://static.docs.arm.com/genc009492/c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf` (visited on 09/04/2020) (cit. on pp. 19, 30).

[62] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2016 (cit. on pp. 19, 29).

[63] D. Champagne and R. B. Lee, "Scalable Architectural Support for Trusted Software", in *Proceedings of the International Symposium on High-Performance Computer Architecture*, ser. HPCA, 2010 (cit. on pp. 19, 29).

[64] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing", in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS, 2003 (cit. on pp. 19, 29).

[65] H. M. Deitel, *An introduction to operating systems*. Addison-Wesley Longman Publishing Co., Inc., 1990 (cit. on p. 21).

[66] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX", in *Proceedings of the International Middleware Conference*, ser. Middleware, 2016 (cit. on p. 21).

[67] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX", in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2016 (cit. on pp. 21, 41, 42, 79, 111).

[68] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves", in *Proceedings of the Annual International Symposium on Computer Architecture*, ser. ISCA, 2017 (cit. on p. 21).

[69] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in Distributed Systems: Theory and Practice", *ACM Transactions on Computer Systems*, 1992 (cit. on p. 23).

[70] NSA, "Tempest: A Signal Problem", *Cryptologic Spectrum*, 1972 (cit. on p. 24).

[71] B. B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical", in *Proceedings of the European Symposium on Research in Computer Security*, ser. ESORICS, 2011 (cit. on p. 24).

[72] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2014 (cit. on p. 24).

[73] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2015 (cit. on p. 24).

[74] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES", in *Proceedings of the Cryptographers' Track at the RSA Conference*, ser. CT-RSA, 2006 (cit. on p. 24).

[75] D. Genkin, I. Pipman, and E. Tromer, "Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs", *Journal of Cryptographic Engineering*, 2015 (cit. on p. 24).

[76] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Page Faults from Telling Your Secrets", in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS, 2016 (cit. on pp. 25, 53).

[77] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical", in *Proceedings of the USENIX Conference on Offensive Technologies*, ser. WOOT, 2017 (cit. on pp. 25, 53).

[78] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX Amplifies the Power of Cache Attacks", in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*, ser. CHES, 2017 (cit. on pp. 25, 53).

[79] F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks", *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018 (cit. on p. 25).

[80] R. Sinha, S. Rajamani, and S. A. Seshia, "A Compiler and Verifier for Page Access Oblivious Computation", in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE, 2017 (cit. on p. 25).

[81] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2017 (cit. on p. 25).

[82] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2017 (cit. on p. 25).

[83] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu", in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS, 2017 (cit. on p. 25).

[84] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2017 (cit. on p. 25).

[85] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization", in *Proceedings of the Annual Computer Security Applications Conference*, ser. ACSAC, 2019 (cit. on p. 25).

[86] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization", in *Proceedings of the Workshop on System Software for Trusted Execution*, ser. SysTEX, 2018 (cit. on p. 25).

[87] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2007 (cit. on p. 26).

[88] J. Laski and W. Stanley, *Software Verification and Analysis: An Integrated, Hands-On Approach*. Springer Science & Business Media, 2009 (cit. on p. 26).

[89] J. Rutkowska. (Sep. 2013). Thoughts on Intel's upcoming Software Guard Extensions (Part 2), [Online]. Available: `https://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html` (visited on 09/04/2020) (cit. on p. 26).

[90] S. Davenport. (Jan. 2014). SGX: the good, the bad and the downright ugly, [Online]. Available: `https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly` (visited on 09/04/2020) (cit. on p. 26).

[91] Intel Corporation. (2016). Intel Software Guard Extensions Commercial Licensing FAQ, [Online]. Available: `https://software.intel.com/content/www/us/en/develop/articles/intel-software-guard-extensions-product-licensing-faq.html` (visited on 09/04/2020) (cit. on p. 27).

[92] R. Lakshmanan. (Aug. 2020). Intel, ARM, IBM, AMD Processors Vulnerable to New Side-Channel Attacks, [Online]. Available: `https://thehackernews.com/2020/08/foreshadow-processor-vulnerability.html` (visited on 09/04/2020) (cit. on p. 27).

[93] Trend Micro. (May 2019). Side-Channel Attacks RIDL, Fallout, and ZombieLoad Affects Millions of Vulnerable Intel Processors, [Online]. Available: `https://www.trendmicro.com/vinfo/de/security/news/vulnerabilities-and-exploits/side-channel-attacks-ridl-fallout-and-zombieload-affects-millions-of-vulnerable-intel-processors` (visited on 09/04/2020) (cit. on p. 27).

[94] M. Kumar. (Jan. 2020). New 'CacheOut' Attack Leaks Data from Intel CPUs, VMs and SGX Enclave, [Online]. Available: `https://thehackernews.com/2020/01/new-cacheout-attack-leaks-data-from.html` (visited on 09/04/2020) (cit. on p. 27).

[95] C. Cimpanu. (Dec. 2019). New Plundervolt attack impacts Intel CPUs, [Online]. Available: `https://www.zdnet.com/article/new-plundervolt-attack-impacts-intel-cpus/` (visited on 09/04/2020) (cit. on p. 27).

[96] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, "Spectre Attacks: Exploiting Speculative Execution", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2019 (cit. on p. 28).

[97] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2018 (cit. on p. 28).

[98] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. (2020). CacheOut: Leaking Data on Intel CPUs via Cache Evictions, [Online]. Available: `https://cacheoutattack.com/` (visited on 09/04/2020) (cit. on p. 28).

[99] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom. (2020). SGAxe: How SGX Fails in Practice, [Online]. Available: `https://sgaxe.com/` (visited on 09/04/2020) (cit. on p. 28).

[100] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2020 (cit. on p. 28).

[101] Amazon.com, Inc. (2020). AWS CloudHSM, [Online]. Available: `https://aws.amazon.com/cloudhsm/` (visited on 09/04/2020) (cit. on p. 29).

[102] S. W. Smith and S. Weingart, "Building a high-performance, programmable secure coprocessor", *Computer Networks*, 1999 (cit. on p. 29).

[103] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. Van Doorn, and S. W. Smith, "Building the IBM 4758 Secure Coprocessor", *Computer*, 2001 (cit. on p. 29).

[104] Trusted Computing Group. (2013). TPM 2.0 Library Specification, [Online]. Available: `https://trustedcomputinggroup.org/resource/tpm-library-specification/` (visited on 09/04/2020) (cit. on p. 29).

[105] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Elsevier, 2004 (cit. on p. 29).

[106] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey", *ACM Computing Surveys*, 2019 (cit. on p. 30).

[107] Advanced Micro Devices, Inc. (2020). AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, [Online]. Available: `https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf` (visited on 09/04/2020) (cit. on pp. 30, 31).

[108] D. Kaplan. (2017). Protecting VM Register State With SEV-ES, [Online]. Available: `https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf` (visited on 09/04/2020) (cit. on p. 31).

[109] IBM Corporation. (2020). IBM Secure Execution for Linux, [Online]. Available: `https://www.ibm.com/downloads/cas/O158MBWG` (visited on 09/04/2020) (cit. on p. 31).

[110] IBM Corporation. (2020). Introducing IBM Secure Execution for Linux 1.1.0, [Online]. Available: `https://public.dhe.ibm.com/software/dw/linux390/docu/l151se00.pdf` (visited on 09/04/2020) (cit. on p. 31).

[111] A. C. Yao, "Protocols for Secure Computations", in *Proceedings of the Annual Symposium on Foundations of Computer Science*, ser. SFCS, 1982 (cit. on p. 34).

[112] A. C.-C. Yao, "How to Generate and Exchange Secrets", in *Proceedings of the Annual Symposium on Foundations of Computer Science*, ser. SFCS, 1986, pp. 162–167 (cit. on p. 34).

[113] O. Goldreich, S. Micali, and A. Wigderson, "How to Play Any Mental Game", in *Proceedings of the Annual ACM Symposium on Theory of Computing*, ser. STOC, 1987 (cit. on pp. 34, 49).

[114] R. Canetti, "Security and Composition of Multi-party Cryptographic Protocols", *Journal of Cryptology*, 2000 (cit. on p. 34).

[115] D. Beaver, S. Micali, and P. Rogaway, "The Round Complexity of Secure Protocols", in *Proceedings of the annual ACM Symposium on Theory of Computing*, ser. STOC, 1990 (cit. on p. 34).

[116] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure Two-Party Computation Is Practical", in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT, 2009 (cit. on p. 34).

[117] U. Feige, J. Killian, and M. Naor, "A Minimal Model for Secure Computation", in *Proceedings of the annual ACM Symposium on Theory of Computing*, ser. STOC, 1994 (cit. on p. 34).

[118] S. Kamara, P. Mohassel, and M. Raykova, "Outsourcing Multi-Party Computation", IACR Cryptology ePrint Archive, Report 2011/272, 2011 (cit. on p. 34).

[119] I. Damgård and Y. Ishai, "Constant-Round Multiparty Computation Using a Black-Box Pseudorandom Generator", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2005 (cit. on p. 34).

[120] D. Evans, V. Kolesnikov, and M. Rosulek, "A Pragmatic Introduction to Secure Multi-Party Computation", *Foundations and Trendsin Privacy and Security*, 2017 (cit. on p. 34).

[121] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu, "Two Can Keep a Secret: A Distributed Architecture for Secure Database Services", in *Proceedings of the Biennial Conference on Innovative Data Systems Research*, ser. CIDR, 2005 (cit. on p. 35).

[122] V. Ciriani, S. D. C. D. Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Combining Fragmentation and Encryption to Protect Privacy in Data Storage", *ACM Transactions on Information and System Security*, TISSEC, 2010 (cit. on p. 35).

[123] S. D. C. di Vimercati, S. Foresti, S. Jajodia, G. Livraga, S. Paraboschi, and P. Samarati, "Fragmentation in Presence of Data Dependencies", *IEEE Transactions on Dependable and Secure Computing*, TDSC, 2014 (cit. on p. 35).

[124] J. Köhler and K. Jünemann, "Securus: From Confidentiality and Access Requirements to Data Outsourcing Solutions", in *Proceedings of the Privacy and Identity Management for Emerging Services and Technologies*, 2014 (cit. on p. 35).

[125] M. Bellare, M. Fischlin, A. O'Neill, and T. Ristenpart, "Deterministic Encryption: Definitional Equivalences and Constructions without Random Oracles", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2008 (cit. on pp. 36, 73).

[126] M. Bellare, T. Kohno, and C. Namprempre, "Authenticated Encryption in SSH: Provably Fixing the SSH Binary Packet Protocol", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2002 (cit. on p. 36).

[127] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2011 (cit. on pp. 37, 73, 80, 91).

[128] R. A. Popa, F. H. Li, and N. Zeldovich, "An ideal-security protocol for order-preserving encoding", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2013 (cit. on p. 37).

[129] F. Kerschbaum and A. Schröpfer, "Optimal Average-Complexity Ideal-Security Order-Preserving Encryption", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2014 (cit. on pp. 37, 73, 80).

[130] F. Kerschbaum, "Frequency-Hiding Order-Preserving Encryption", in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2015 (cit. on pp. 37, 38, 91).

[131] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti, "Modular Order-Preserving Encryption, Revisited", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2015 (cit. on pp. 37, 91).

[132] D. Boneh, K. Lewi, M. Raykova, A. Sahai, M. Zhandry, and J. Zimmerman, "Semantically Secure Order-Revealing Encryption: Multi-input Functional Encryption Without Obfuscation", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2015 (cit. on pp. 37, 41, 73, 74, 91).

[133] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu, "Practical Order-Revealing Encryption with Limited Leakage", in *Proceedings of the International Conference on Fast Software Encryption*, ser. FSE, 2016 (cit. on pp. 37, 73).

[134] K. Lewi and D. J. Wu, "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds", in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016 (cit. on pp. 37, 41).

[135] F. B. Durak, T. M. DuBuisson, and D. Cash, "What Else is Revealed by Order-Revealing Encryption?", in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016 (cit. on p. 37).

[136] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic Attacks on Secure Outsourced Databases", in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2016 (cit. on pp. 38, 91).

[137] P. Grubbs, M. Lacharite, B. Minaud, and K. G. Paterson, "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2018 (cit. on pp. 38, 91).

[138] Z. Gui, O. Johnson, and B. Warinschi, "Encrypted Databases: New Volume Attacks against Range Queries", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2019 (cit. on pp. 38, 91).

[139] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A Survey of Provably Secure Searchable Encryption", *ACM Computing Surveys*, 2014 (cit. on p. 38).

[140] D. X. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2000 (cit. on pp. 38, 40, 49).

[141] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public Key Encryption with Keyword Search", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2004 (cit. on pp. 38, 40).

[142] E. Goh, "Secure Indexes", IACR Cryptology ePrint Archive, Report 2003/216, 2003 (cit. on p. 39).

[143] Y.-C. Chang and M. Mitzenmacher, "Privacy Preserving Keyword Searches on Remote Encrypted Data", in *Proceedings of the International Conference on Applied Cryptography and Network Security*, ser. ACNS, 2005 (cit. on p. 39).

[144] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2006 (cit. on pp. 39, 46, 49).

[145] D. Boneh and B. Waters, "Conjunctive, Subset, and Range Queries on Encrypted Data", in *Proceedings of the Theory of Cryptography Conference*, ser. TCC, 2007 (cit. on pp. 39, 49, 55, 81).

[146] E. Shen, E. Shi, and B. Waters, "Predicate Privacy in Encryption Systems", in *Proceedings of the Theory of Cryptography Conference*, ser. TCC, 2009 (cit. on pp. 39, 49, 55).

[147] E. Shi, J. Bethencourt, H. T.-H. Chan, D. X. Song, and A. Perrig, "Multi-Dimensional Range Query over Encrypted Data", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2007 (cit. on pp. 39, 49, 55, 81).

[148] Y. Lu, "Privacy-Preserving Logarithmic-time Search on Encrypted Data in Cloud", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2012 (cit. on pp. 39, 49, 55, 74, 81, 91).

[149] I. Demertzis, S. Papadopoulos, O. Papapetrou, A. Deligiannakis, and M. Garofalakis, "Practical Private Range Search Revisited", in *Proceedings of the International Conference on Management of Data*, ser. SIGMOD, 2016 (cit. on pp. 39, 49, 55, 69, 71, 72, 81).

[150] S. Halevi and V. Shoup, "Bootstrapping for HElib", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2015 (cit. on p. 40).

[151] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds", in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT, 2016 (cit. on p. 40).

[152] S. Goldwasser and S. Micali, "Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information", in *Proceedings of the ACM Symposium on the Theory of Computing*, ser. STOC, 1982 (cit. on p. 40).

[153] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 1999 (cit. on p. 40).

[154] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Transactions on Information Theory*, 1985 (cit. on p. 40).

[155] A. Sahai and B. Waters. (2008). Slides on functional encryption. PowerPoint presentation, [Online]. Available: `https://www.cs.utexas.edu/~bwaters/presentations/files/functional.ppt` (visited on 09/04/2020) (cit. on p. 40).

[156] A. O'Neill, "Definitional Issues in Functional Encryption", IACR Cryptology ePrint Archive, Report2010/556, 2010 (cit. on p. 40).

[157] D. Boneh, A. Sahai, and B. Waters, "Functional Encryption: Definitions and Challenges", in *Proceedings of the Conference on Theory of Cryptography*, ser. TCC, 2011 (cit. on pp. 40, 49).

[158] S. Goldwasser, S. D. Gordon, V. Goyal, A. Jain, J. Katz, F.-H. Liu, A. Sahai, E. Shi, and H.-S. Zhou, "Multi-input Functional Encryption", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2014 (cit. on p. 40).

[159] A. Lewko, T. Okamoto, A. Sahai, K. Takashima, and B. Waters, "Fully Secure Functional Encryption: Attribute-Based Encryption and (Hierarchical) Inner Product Encryption", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2010 (cit. on p. 40).

[160] A. Sahai and B. Waters, "Fuzzy Identity-Based Encryption", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2005 (cit. on pp. 40, 98, 101).

[161] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-Based Encryption for Fine-grained Access Control of Encrypted Data", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2006 (cit. on pp. 40, 98, 101).

[162] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable Garbled Circuits and Succinct Functional Encryption", in *Proceedings of the Annual ACM Symposium on Theory of Computing*, ser. STOC, 2013 (cit. on p. 41).

[163] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits", *SIAM Journal on Computing*, 2016 (cit. on p. 41).

[164] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven", in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2014 (cit. on pp. 41, 73, 79).

[165] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the Library OS from the Top Down", in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2011 (cit. on p. 41).

[166] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX", in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC, 2017 (cit. on p. 41).

[167] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications", in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys, 2014 (cit. on p. 41).

[168] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux Applications With SGX Enclaves", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2017 (cit. on p. 42).

[169] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "Orthogonal Security with Cipherbase", in *Proceedings of the Biennial Conference on Innovative Data Systems Research*, ser. CIDR, 2013 (cit. on pp. 42, 43).

[170] A. Arasu, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, and R. Ramamurthy, "Transaction Processing on Confidential Data using Cipherbase", in *Proceedings of the IEEE International Conference on Data Engineering*, ser. ICDE, 2015 (cit. on pp. 43, 80).

[171] Intel Corporation. (2020). Library for Intel Software Guard Extensions, [Online]. Available: `https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/academic-research.html` (visited on 09/04/2020) (cit. on p. 43).

[172] Microsoft. (2020). Cloud Computing Services — Microsoft Azure, [Online]. Available: `https://azure.microsoft.com/` (visited on 09/04/2020) (cit. on pp. 47, 93, 113).

[173] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices", in *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET, 1970 (cit. on p. 49).

[174] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd. McGraw-Hill Higher Education, 2002 (cit. on p. 49).

[175] Oracle Corporation. (2020). 8.3.9 Comparison of B-Tree and Hash Indexes, [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/index-btree-hash.html` (visited on 09/04/2020) (cit. on p. 49).

[176] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide*. O'Reilly and Associates, 2010 (cit. on p. 49).

[177] D. Giampaolo, *Practical file system design with the Be file system*. Morgan Kaufmann Publishers Inc., 1998 (cit. on p. 49).

[178] Microsoft. (Jan. 2012). Building the next generation file system for Windows: ReFS, [Online]. Available: `https://docs.microsoft.com/en-us/archive/blogs/b8/building-the-next-generation-file-system-for-windows-refs` (visited on 09/04/2020) (cit. on p. 49).

[179] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation", in *Proceedings of the Annual ACM Symposium on Theory of Computing*, ser. STOC, 1988 (cit. on p. 49).

[180] J. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully Homomorphic Encryption over the Integers with Shorter Public Keys", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2011 (cit. on pp. 49, 73).

[181] S. Kamara and T. Moataz, "SQL on Structurally-Encrypted Databases", in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT, 2016 (cit. on p. 49).

[182] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing", in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011 (cit. on pp. 49, 54, 73, 80).

[183] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner, "Rich Queries on Encrypted Data: Beyond Exact Matches", in *Proceedings of the European Symposium on Research in Computer Security*, ser. ESORICS, 2015 (cit. on pp. 49, 55, 69, 71, 80).

[184] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2015 (cit. on p. 54).

[185] O. Ohrimenko, F. Schuster, C. Fournet, A. Meht, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2016 (cit. on p. 54).

[186] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing Analytical Queries over Encrypted Data", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2013 (cit. on pp. 54, 73, 80).

[187] B. Hore, S. Mehrotra, and G. Tsudik, "A Privacy-Preserving Index for Range Queries", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2004 (cit. on p. 54).

[188] P. Wang and C. Ravishankar, "Secure and efficient range queries on outsourced databases using Rp-trees", in *Proceedings of the IEEE International Conference on Data Engineering*, ser. ICDE, 2013 (cit. on p. 54).

[189] J. Li and E. R. Omiecinski, "Efficiency and Security Trade-off in Supporting Range Queries on Encrypted Databases", in *Proceedings of the Conference on Data and Applications Security*, ser. DBSec, 2005 (cit. on p. 54).

[190] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind Seer: A Scalable Private DBMS", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2014 (cit. on pp. 54, 80).

[191] M. Egorov and M. Wilkison, "ZeroDB white paper", arXiv.org, arXiv:1602.07168, 2016 (cit. on pp. 54, 80).

[192] R. Poddar, T. Boelter, and R. A. Popa, "Arx: An Encrypted Database using Semantically Secure Encryption", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2019 (cit. on p. 54).

[193] B. Wang, Y. Hou, M. Li, H. Wang, and H. Li, "Maple: Scalable Multi-Dimensional Range Search over Encrypted Cloud Data with Tree-based Index", in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS, 2014 (cit. on p. 55).

[194] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Roşu, and M. Steiner, "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2013 (cit. on pp. 55, 71, 80).

[195] M. Naveed, "The Fallacy of Composition of Oblivious RAM and Searchable Encryption", IACR Cryptology ePrint Archive, Report 2015/668, 2015 (cit. on p. 55).

[196] S. Garg, P. Mohassel, and C. Papamanthou, "TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2016 (cit. on p. 55).

[197] D. J. Bernstein, T. Lange, and P. Schwabe, "The Security Impact of a New Cryptographic Library", in *Proceedings of the International Conference on Cryptology and Information Security in Latin America*, ser. LATINCRYPT, 2012 (cit. on p. 64).

[198] L. Xu. (2010). Securing the Enterprise with Intel AES-NI, [Online]. Available: `https://www.intel.de/content/dam/doc/white-paper/enterprise-security-aes-ni-white-paper.pdf` (visited on 09/04/2020) (cit. on p. 64).

[199] K. Mowery, S. Keelveedhi, and H. Shacham, "Are AES x86 Cache Timing Attacks Still Feasible?", in *Proceedings of the ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW, 2012 (cit. on p. 64).

[200] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating Compression and Execution in Column-Oriented Database Systems", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2006 (cit. on pp. 73, 76).

[201] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units", *Proceedings of the VLDB Endowment*, 2009 (cit. on pp. 73, 76).

[202] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The Design and Implementation of Modern Column-Oriented Database Systems", *Foundations and Trends® in Databases*, 2013 (cit. on pp. 73, 76).

[203] P. A. Boncz and M. L. Kersten, "MIL primitives for querying a fragmented world", *The VLDB Journal*, 1999 (cit. on pp. 73, 76).

[204] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 1985 (cit. on pp. 73, 76).

[205] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: A Column-Oriented DBMS", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2005 (cit. on pp. 73, 76, 88, 89).

[206] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation Techniques for Main Memory Database Systems", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 1984 (cit. on pp. 73, 75).

[207] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on knowledge and data engineering*, 1992 (cit. on pp. 73, 75).

[208] P.-Å. Larson and J. Levandoski, "Modern Main-Memory Database Systems", *Proceedings of the VLDB Endowment*, 2016 (cit. on pp. 73, 75).

[209] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A Secure Database using SGX", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2018 (cit. on pp. 73, 79, 80).

[210] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the Memory Wall in MonetDB", *Communications of the ACM*, 2008 (cit. on pp. 74, 90).

[211] P. A. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution", in *Proceedings of the Conference on Innovative Data Systems*, ser. CIDR, 2005 (cit. on pp. 74, 90).

[212] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures", *IEEE Data Engineering Bulletin*, 2012 (cit. on pp. 74, 90).

[213] I. Müller, C. Ratsch, and F. Faerber, "Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems", in *Proceedings of the International Conference on Extending Database Technology*, ser. EDBT, 2014 (cit. on p. 75).

[214] C. Lemke, K. Sattler, F. Faerber, and A. Zeier, "Speeding Up Queries in Column Stores", in *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, ser. DaWaK, 2010 (cit. on p. 75).

[215] SAP America, Inc. (2020). SAP HANA — In-Memory Database, [Online]. Available: `https://www.sap.com/products/hana.html` (visited on 09/04/2020) (cit. on p. 75).

[216] Oracle. (2020). In-Memory Database — Oracle, [Online]. Available: `https://www.oracle.com/database/technologies/in-memory.html` (visited on 09/04/2020) (cit. on p. 75).

[217] MonetDB B.V. (2020). Home — MonetDB, [Online]. Available: `https://www.monetdb.org/` (visited on 09/04/2020) (cit. on p. 75).

[218] J. Krueger, M. Grund, A. Zeier, and H. Plattner, "Enterprise Application-specific Data Management", in *Proceedings of the IEEE International Enterprise Distributed Object Computing Conference*, ser. EDOC, 2010 (cit. on p. 76).

[219] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden, "Performance Tradeoffs in Read-Optimized Databases", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2006 (cit. on pp. 76, 88).

[220] F. Hahn and F. Kerschbaum, "Poly-Logarithmic Range Queries on Encrypted Data with Small Leakage", in *Proceedings of the ACM on Cloud Computing Security Workshop*, ser. CCSW, 2016 (cit. on p. 81).

[221] F. Kerschbaum and A. Tueno, "An Efficiently Searchable Encrypted Data Structure for Range Queries", in *Proceedings of the European Symposium on Research in Computer Security*, ser. ESORICS, 2019 (cit. on pp. 83, 91).

[222] D. Pouliot, S. Griffy, and C. V. Wright, "The Strength of Weak Randomization: Efficiently Searchable Encryption with Minimal Leakage", IACR Cryptology ePrint Archive, Report 2017/1098, 2017 (cit. on pp. 86, 92).

[223] S. Krastnikov, F. Kerschbaum, and D. Stebila, "Efficient Oblivious Database Joins", arXiv.org, arXiv:2003.09481, 2020 (cit. on p. 88).

[224] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database – An Architecture Overview", *IEEE Data Engineering Bulletin*, 2012 (cit. on p. 89).

[225] F. Hübner, J. Böse, J. Krüger, C. Tosun, A. Zeier, and H. Plattner, "A cost-aware strategy for merging differential stores in column-oriented in-memory DBMS", in *Proceedings of the International Workshop on Business Intelligence for the Real-Time Enterprise*, ser. BIRTE, 2011 (cit. on p. 89).

[226] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace: Oblivious Memory Primitives from Intel SGX", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2018 (cit. on pp. 89, 92).

[227] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform", in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI, 2017 (cit. on pp. 89, 92).

[228] M. McCutchen. (2014). C++ Big Integer Library, [Online]. Available: `https://mattmccutchen.net/bigint/` (visited on 09/04/2020) (cit. on p. 90).

[229] V. Bindschaedler, P. Grubbs, D. Cash, T. Ristenpart, and V. Shmatikov, "The Tao of Inference in Privacy-Protected Databases", in *Proceedings of the International Conference on Very Large Data Bases*, ser. VLDB, 2018 (cit. on p. 91).

[230] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Réveillère, "Anonymous and Confidential File Sharing over Untrusted Clouds", in *Proceedings of the Symposium on Reliable Distributed Systems*, ser. SRDS, 2019 (cit. on pp. 97, 98, 102).

[231] MEGA. (2020). MEGA, [Online]. Available: `https://mega.nz` (visited on 09/04/2020) (cit. on p. 97).

[232] Sync.com Inc. (2020). Sync Secure Cloud Storage—Privacy Guaranteed, [Online]. Available: `https://www.sync.com` (visited on 09/04/2020) (cit. on p. 97).

[233] E. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing Remote Untrusted Storage", in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2003 (cit. on pp. 97, 98, 102).

[234] A. Shamir and D. Chaum, "Identity-Based Cryptosystems and Signature Schemes", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 1984 (cit. on pp. 98, 101).

[235] D. Boneh and M. Franklin, "Identity-Based Encryption from the Weil Pairing", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2001 (cit. on pp. 98, 101).

[236] A. Fiat and M. Naor, "Broadcast Encryption", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 1993 (cit. on pp. 98, 101).

[237] D. Boneh, C. Gentry, and B. Waters, "Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2005 (cit. on pp. 98, 101, 102).

[238] C. Delerablée, "Identity-Based Broadcast Encryption with Constant Size Ciphertexts and Private Keys", in *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT, 2007 (cit. on pp. 98, 102).

[239] R. Sakai and J. Furukawa, "Identity-Based Broadcast Encryption", IACR Cryptology ePrint Archive, Report 2007/217, 2007 (cit. on pp. 98, 102).

[240] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage", in *Proceedings of the USENIX conference on file and storage technologies*, ser. FAST, 2003 (cit. on pp. 98, 102).

[241] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof", in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC, 2011 (cit. on pp. 98, 102).

[242] K. E. Fu, "Group sharing and random access in cryptographic storage file systems", Master's thesis, Massachusetts Institute of Technology, 1999 (cit. on p. 98).

[243] J. Li, C. Qin, P. P. Lee, and J. Li, "Rekeying for Encrypted Deduplication Storage", in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2016 (cit. on pp. 98, 102, 103).

[244] W. C. Garrison, A. Shull, S. Myers, and A. J. Lee, "On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies in the Cloud", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2016 (cit. on pp. 98, 102, 103).

[245] S. Contiu, R. Pires, S. Vaucher, M. Pasin, P. Felber, and L. Réveillère, "IBBE-SGX: Cryptographic Group Access Control using Trusted Execution Environments", in *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2018 (cit. on pp. 98, 102, 103).

[246] J. B. Djoko, J. Lange, and A. J. Lee, "NEXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX", in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2019 (cit. on pp. 98, 102, 103).

[247] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "PESOS: Policy Enhanced Secure Object Store", in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys, 2018 (cit. on pp. 98, 102, 103).

[248] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015 (cit. on p. 99).

[249] B. Waters, "Efficient Identity-Based Encryption Without Random Oracles", in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT, 2005 (cit. on p. 101).

[250] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-Policy Attribute-Based Encryption", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2007 (cit. on p. 101).

[251] D. Naor, M. Naor, and J. Lotspiech, "Revocation and Tracing Schemes for Stateless Receivers", in *Proceedings of the Annual International Cryptology Conference*, ser. CRYPTO, 2001 (cit. on p. 101).

[252] K. He, J. Weng, J. Liu, J. K. Liu, W. Liu, and R. H. Deng, "Anonymous Identity-Based Broadcast Encryption with Chosen-Ciphertext Security", in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS, 2016 (cit. on p. 102).

[253] A. Boldyreva, V. Goyal, and V. Kumar, "Identity-based Encryption with Efficient Revocation", in *Proceedings of the ACM Conference on Computer and Communications Security*, ser. CCS, 2008 (cit. on p. 103).

[254] M. Green and G. Ateniese, "Identity-Based Proxy Re-encryption", in *Proceedings of the International Conference on Applied Cryptography and Network Security*, ser. ACNS, 2007 (cit. on p. 103).

[255] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming Space from Duplicate Files in a Serverless Distributed File System", in *Proceedings of the International Conference on Distributed Computing Systems*, ser. ICDCS, 2002 (cit. on p. 107).

[256] S. Keelveedhi, M. Bellare, and T. Ristenpart, "DupLESS: Server-Aided Encryption for Deduplicated Storage", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2013 (cit. on p. 107).

[257] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage", in *Proceedings of the USENIX conference on file and storage technologies*, ser. FAST, 2002 (cit. on p. 107).

[258] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System", in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP, 2001 (cit. on p. 107).

[259] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services, the case of deduplication in cloud storage", in *Proceedings of the Symposium on Security and Privacy*, ser. S&P, 2010 (cit. on p. 108).

[260] F. Armknecht, C. Boyd, G. T. Davies, K. Gjøsteen, and M. Toorani, "Side Channels in Deduplication: Trade-offs between Leakage and Efficiency", in *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, ser. ASIACCS, 2017 (cit. on p. 108).

[261] H. Ritzdorf, G. Karame, C. Soriente, and S. Čapkun, "On Information Leakage in Deduplicated Storage Systems", in *Proceedings of the ACM on Cloud Computing Security Workshop*, ser. CCSW, 2016 (cit. on p. 108).

[262] git-scm.com. (2020). 10.2 Git Internals - Git Objects, [Online]. Available: `https://git-scm.com/book/en/v2/Git-Internals-Git-Objects` (visited on 09/04/2020) (cit. on p. 108).

[263] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback Protection for Trusted Execution", in *Proceedings of the USENIX Security Symposium*, ser. USENIX Security, 2017 (cit. on p. 110).

[264] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory", in *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN, 2017 (cit. on p. 110).

[265] E. J. Whitehead and M. Wiggins, "WebDAV: IEFT Standard for Collaborative Authoring on the Web", 1998 (cit. on p. 111).

[266] Cx File Explorer. (2020). Cx File Explorer, [Online]. Available: `https://play.google.com/store/apps/details?id=com.cxinventor.file.explorer` (visited on 09/04/2020) (cit. on p. 111).

[267] Schimera Pty Ltd. (2020). WebDAV Navigator, [Online]. Available: `https://apps.apple.com/de/app/webdav-navigator/id382551345` (visited on 09/04/2020) (cit. on p. 111).

[268] South River Technologies, Inc. (2020). WebDAV Client for Windows and Mac — WebDrive, [Online]. Available: `https://webdrive.com/webdav-with-webdrive/` (visited on 09/04/2020) (cit. on p. 111).

[269] NetDocuments Software Inc. (2020). WebDAV on Mac OS X, [Online]. Available: `https://support.netdocuments.com/hc/en-us/articles/205218800` (visited on 09/04/2020) (cit. on p. 111).

[270] Free Software Foundation, Inc. (2020). davfs2 - Summary, [Online]. Available: `https://savannah.nongnu.org/projects/davfs2` (visited on 09/04/2020) (cit. on p. 111).

[271] Joyent, Inc. and other Node contributors. (2020). HTTP Parser, [Online]. Available: `https://github.com/nodejs/http-parser` (visited on 09/04/2020) (cit. on p. 111).

[272] F. Zhang. (2019). mbedtls-SGX: a TLS stack in SGX, [Online]. Available: `https://github.com/bl4ck5un/mbedtls-SGX` (visited on 09/04/2020) (cit. on p. 111).

[273] wolfSSL Inc. (Jan. 2017). wolfSSL with Intel SGX, [Online]. Available: `https://www.wolfssl.com/wolfssl-with-intel-sgx/` (visited on 09/04/2020) (cit. on p. 111).

[274] The OpenSSL Project. (2020). OpenSSL, [Online]. Available: `https://github.com/openssl/openssl` (visited on 09/04/2020) (cit. on p. 111).

[275] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with Verified Cryptographic Security", in *Proceedings of the IEEE Symposium on Security and Privacy*, ser. S&P, 2013 (cit. on p. 112).

[276] The Apache Software Foundation. (2020). Welcome! The Apache HTTP Server Project, [Online]. Available: `https://httpd.apache.org/` (visited on 09/04/2020) (cit. on p. 113).

[277] nginx. (2020). nginx news, [Online]. Available: `https://nginx.org/` (visited on 09/04/2020) (cit. on p. 113).