BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

SECURE DATA SHARING PROTOCOL FOR PEER-TO-PEER SYSTEMS

PROTOKOL PRO BEZPEČNÉ SDÍLENÍ DAT V PEER-TO-PEER SYSTÉMECH

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PROKOP SCHIELD**
AUTOR PRÁCE

SUPERVISOR **Ing. JIŘÍ HYNEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

164375

Institut:	Department of Information Systems (DIFS)
Student:	**Schield Prokop**
Programme:	Information Technology
Title:	**Secure Data Sharing Protocol for Peer-to-Peer Systems**
Category:	Information Systems
Academic year:	2024/25

Assignment:

1. Study secure encrypted data storage systems (e.g., LUKS), peer-to-peer data sharing systems (e.g., BitTorrent), and distributed file systems (e.g., IPFS).
2. Review symmetric block encryption algorithms, comparing performance and security.
3. Analyze use cases for secure data sharing and define key requirements.
4. Design a secure data serialization format, ensuring encryption and data integrity.
5. Design a protocol for secure peer-to-peer data sharing.
6. Implement the designed protocol and peers.
7. Test and evaluate the implementation for security and performance.

Literature:

- Oram, A. (2001). Peer-to-Peer: Harnessing the Power of Disruptive Technologies. United States of America: O'Reilly Media.
- Schneier, B. (2017). Applied Cryptography: Protocols, Algorithms and Source Code in C. Germany: Wiley.
- Stallings, W. (2017). Cryptography and network security. United Kingdom: Pearson Prentice Hall.
- Tanenbaum, A. S., Steen, M. V. (2023). *Distributed Systems*. 4th Edition. distributed-systems.net.

Requirements for the semestral defence:
Items 1 to 5.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Hynek Jiří, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	21.5.2025
Approval date:	22.10.2024

Abstract

This bachelor's thesis confronts the challenge of secure data sharing within decentralized peer-to-peer networks by proposing `ScatterNet`, a protocol engineered to ensure data confidentiality, integrity, availability, and resilience. Core contributions include a robust data serialization format and a minimal communication interface. The serialization format defines the `DataChunk`, identified by a cryptographic `Hash` constructed from SHA-256 and BLAKE3 outputs, `PackedInt`-encoded length, and Reed-Solomon parity. Data protection employs Deflate compression, ChaCha20-Poly1305 AEAD encryption—keyed by a hash of the serialized data with a key-derived nonce—and a custom Long Error Correction Code (Long ECC) featuring a self-correcting header and interleaved Reed-Solomon protection for the encrypted payload. A versatile `HKey` identifier facilitates data addressing. The communication protocol, operating between Ed25519-identified peers, specifies `FETCH` and `PUT` operations and incorporates a local reputation system. A modular Rust reference implementation utilizing iroh for QUIC-based peer-to-peer communication demonstrates the protocol's viability, presenting a comprehensive solution for secure, resilient data exchange.

Abstrakt

Tato bakalářská práce řeší problematiku bezpečného sdílení dat v decentralizovaných peer-to-peer sítích návrhem protokolu `ScatterNet`, vyvinutého k zajištění důvěrnosti, integrity, dostupnosti a odolnosti dat. Klíčové přínosy zahrnují robustní formát serializace dat a minimální komunikační rozhraní. Serializační formát definuje `DataChunk`, identifikovaný kryptografickým otiskem `Hash` konstruovaným z výstupů SHA-256 a BLAKE3, délky kódované formátem `PackedInt` a Reed-Solomonovy parity. Ochrana dat využívá kompresi Deflate, AEAD šifrování ChaCha20-Poly1305 – s klíčem odvozeným z hashe serializovaných dat a nonce odvozenou z klíče – a vlastní schéma Long Error Correction Code (Long ECC) se samoopravnou hlavičkou a prokládanou Reed-Solomonovou ochranou šifrovaného obsahu. Univerzální identifikátor `HKey` umožňuje adresaci dat. Komunikační protokol, operující mezi peery identifikovanými klíči Ed25519, specifikuje operace `FETCH` a `PUT` a zahrnuje lokální systém reputace. Modulární referenční implementace v Rustu, využívající iroh pro peer-to-peer komunikaci přes QUIC, demonstruje životaschopnost protokolu a prezentuje komplexní řešení pro bezpečné a odolné sdílení dat.

Keywords

P2P, peer-to-peer systems, secure data sharing, decentralized systems, cryptographic hash functions, content addressing, authenticated encryption, ChaCha20-Poly1305, data integrity, data confidentiality, Rust, distributed storage, QUIC, iroh, ScatterNet

Klíčová slova

P2P, peer-to-peer systémy, bezpečné sdílení dat, decentralizované systémy, kryptografické hašovací funkce, adresování obsahem, autentizované šifrování, ChaCha20-Poly1305, integrita dat, důvěrnost dat, Rust, distribuovaná úložiště, QUIC, iroh, ScatterNet

Reference

SCHIELD, Prokop. *Secure Data Sharing Protocol for Peer-to-Peer Systems.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Hynek, Ph.D.

Rozšířený abstrakt

Decentralizované peer-to-peer sítě nabízí potenciál pro odolnější a svobodnější sdílení dat, avšak jejich zabezpečení zůstává výzvou. Současné peer-to-peer systémy často upřednostňují efektivní distribuci před komplexní ochranou citlivých informací. Tato bakalářská práce představuje protokol `ScatterNet` – řešení zaměřené na bezpečné sdílení dat v peer-to-peer prostředích. Cílem je vytvořit robustní formát pro serializaci dat a komunikační rozhraní zajišťující důvěrnost, integritu, dostupnost a odolnost sdílených dat.

Východiskem práce je analýza existujících technologií pro šifrované ukládání (např. LUKS), sdílení dat v sítích peer-to-peer (např. BitTorrent) a distribuované souborové systémy (např. IPFS). Práce zkoumá také symetrické šifrovací algoritmy (AES, ChaCha20) a schémata AEAD (AES-GCM, ChaCha20-Poly1305). Na základě této rešerše a analýzy případů užití byly stanoveny klíčové požadavky pro protokol `ScatterNet`: zajištění důvěrnosti, integrity, dostupnosti, odolnosti dat proti poškození a ztrátě, škálovatelnost systému, efektivní využití zdrojů a snadná integrace pro vývojáře.

Bezpečný formát serializace tvoří jádro protokolu `ScatterNet`. Základní datovou jednotku představuje `DataChunk` – struktura obsahující surová data a jejich kryptografický otisk `Hash`. Tento `Hash` je tvořen 64znakovým řetězcem v kódování Base64. Jeho 48bajtová binární podoba vzniká kombinací výstupů funkcí SHA-256 a BLAKE3 pomocí operace XOR, ke kterým se připojí 2bajtová přibližná délka vstupu ve formátu `PackedInt` a následně 14 paritních bajtů Reed-Solomonova kódu RS(48,34) pro zvýšení odolnosti. Výsledný identifikátor `Hash` je vytvořen zakódováním této binární podoby pomocí Base64.

Ochrana dat začíná serializací struktury `DataChunk` (ve formátu `Hash(data) || data`) a její kompresí algoritmem Deflate. Šifrovací klíč `EncryptionKey` je generován jako hodnota `Hash` z nekomprimované serializované podoby `DataChunk`. Komprimovaná data jsou šifrována pomocí schématu ChaCha20-Poly1305, kde jako nonce slouží posledních 12 bajtů z paritní části klíče `EncryptionKey` v binární podobě. Výsledný šifrovaný blok `ChaChaEncrypted` je dále zabezpečen vlastním schématem pro korekci chyb Long Error Correction Code (Long ECC). Toto schéma využívá 16bajtovou hlavičku `LongEccHeader` s metadaty chráněnými kódem RS(16,12). Celý blok je navíc zabezpečen iterativním prokládaným Reed-Solomonovým kódováním s paritou připojenou na konec řetězce. Výsledkem je plně zabezpečený datový blok `EncryptedData`.

Pro adresaci a verifikaci dat slouží univerzální identifikátor `HKey`. Jeho šifrovaná varianta označená znakem 'E' obsahuje hodnotu `Hash` vypočítanou z bloku `EncryptedData` a původní klíč `EncryptionKey`. Existují také další varianty: přímé kódování malých dat (označeno 'B'), odkazy na nešifrovaná data ('D'), odkazy na seznamy identifikátorů `HKey` ('L') a varianty pro skládání dat z více zdrojů ('[' pro sekvenční seznamy a '{' pro strukturované objekty s rozsahy).

Protokol `ScatterNet` definuje způsob komunikace mezi peery, kde každý peer je identifikován pomocí veřejného klíče Ed25519. Jedná se o čistě logický protokol nezávislý na konkrétním síťovém transportu či struktuře paketů. Základní komunikační rozhraní obsahuje operaci `FETCH hash` pro získání dat `EncryptedData` a operaci `PUT data` pro jejich ukládání. Operace `PUT` primárně přijímá již zabezpečená data `EncryptedData` spolu s identifikátorem `HKey`, přičemž úložiště ověřuje správnost hodnoty hash. Volitelně může implementace podporovat i příjem nezpracovaných dat, která sama převede do zabezpečené podoby a vrátí příslušný identifikátor `HKey`. Pro zajištění spravedlivého využívání zdrojů byl navržen lokální systém reputace: úspěšné provedení operace `FETCH` zvyšuje reputaci poskytovatele (+1 bod z pohledu klienta), úspěšné provedení operace `PUT` zvyšuje reputaci

úložiště (+3 body z pohledu klienta) a současně snižuje reputaci klienta (-1 bod z pohledu úložiště). Při nízké reputaci může dojít k odmítnutí požadavků `PUT`.

Referenční implementace protokolu `ScatterNet` je vytvořena v jazyce Rust jako sada modulárních knihoven pokrývajících jednotlivé funkce (`Base64`, `PackedInt`, `Hash`, `HKey`, Deflate, ECC, `DataChunk` a šifrování). Hlavní aplikace pro komunikaci mezi peery s názvem `scatter-net` integruje tyto moduly a pro zajištění komunikace peer-to-peer využívá knihovnu iroh (spolu s QUIC a Tokio). Serializace paketů je implementována pomocí knihovny bincode a komprese dat pomocí knihovny `ps-deflate`.

Tato práce představuje komplexní návrh protokolu pro bezpečné sdílení dat v prostředí peer-to-peer. Návrh zahrnuje robustní formát serializace s důrazem na důvěrnost, integritu a odolnost proti chybám, flexibilní systém identifikátorů `HKey` a minimalistické komunikační rozhraní doplněné o systém reputace. Modulární implementace v jazyce Rust potvrzuje praktickou realizovatelnost navrženého řešení, které vytváří solidní základ pro další rozvoj.

Secure Data Sharing Protocol for Peer-to-Peer Systems

Declaration

I declare that am the sole author of this thesis, which I have written under the supervision of Ing. Jiří Hynek, Ph.D., and that I have cited all sources and publications I have used. The external help I've received was limited to typesetting. I used Generative AI (LLMs) to find relevant and primary sources, to generate and format BibTeX entries, and to translate, transform, style, format, and improve my work.

.....
Prokop Schield
May 21, 2025

Acknowledgements

I would like to thank my supervisor for having so much patience with me.

Contents

List of Figures

Introduction

The proliferation of decentralized technologies has reignited interest in peer-to-peer systems as a paradigm for data sharing and communication. Peer-to-peer architectures offer inherent advantages over centralized models, including enhanced resilience against single points of failure, improved censorship resistance, and greater user autonomy [36]. However, these benefits are often accompanied by significant security challenges, particularly when the data being shared is confidential or sensitive. Ensuring the privacy, integrity, and controlled accessibility of data in an environment where trust is not inherent and participants can be anonymous or ephemeral remains a complex undertaking.

Existing peer-to-peer data sharing solutions, while often excelling in content dissemination efficiency and scalability, frequently provide insufficient guarantees for secure data exchange. Many protocols (e.g. BitTorrent) lack robust end-to-end encryption, secure key management facilities, or fine-grained access control mechanisms, leaving data vulnerable to unauthorized access, modification, or eavesdropping. Similarly, while distributed filesystems offer powerful primitives for decentralized storage, their standard implementations may not seamlessly integrate the comprehensive security features required for private, controlled data sharing among specific sets of peers. This gap highlights the need for a specialized protocol designed explicitly for secure data sharing in peer-to-peer networks.

This thesis addresses these challenges by proposing `ScatterNet`, a novel protocol for secure peer-to-peer data sharing. The primary objective of `ScatterNet` is to provide a robust framework that ensures the confidentiality, integrity, availability, and long-term durability of data exchanged among peers. To achieve this, the work encompasses several key contributions:

- A detailed analysis of existing secure storage solutions, peer-to-peer systems, and distributed filesystems, identifying their strengths and limitations in the context of secure data sharing.
- A comparative review of symmetric encryption algorithms and authenticated encryption schemes to inform the selection of appropriate cryptographic primitives.
- The definition of key requirements for a secure peer-to-peer data sharing system, derived from an analysis of relevant use cases.
- The design of a secure data serialization format that incorporates strong encryption (ChaCha20-Poly1305), robust content-addressable hashing (combining SHA-256, BLAKE3, and Reed-Solomon parity), error correction codes (Reed-Solomon for both hash identifiers and encrypted payloads), and a versatile data referencing mechanism (`HKey`).

- The specification of a minimalist peer-to-peer communication protocol defining core data exchange operations (`FETCH` and `PUT`) and a local reputation system to foster fair resource utilization.
- An overview of the reference implementation of `ScatterNet` in Rust, detailing its modular architecture.

The `ScatterNet` protocol achieves its security goals through a general abstraction of an arbitrary chunk of data, termed a `DataChunk` (detailed in Section 4.1). Persistent data, in its protected form, is addressed by a unique identifier called an `HKey` (Section 4.5), which encapsulates information necessary for retrieval and decryption.

This thesis is structured as follows: Chapter 1 reviews existing data storage and distribution solutions. Subsequently, Chapter 2 provides an analysis of symmetric encryption algorithms and authenticated encryption. Use cases and key requirements for secure data sharing are discussed in Chapter 3. The design of the `ScatterNet` secure data serialization format is detailed in Chapter 4, while Chapter 5 specifies the peer-to-peer communication protocol. The reference implementation of the proposed system is described in Chapter 6, and its testing and evaluation are presented in Chapter 7.

Chapter 1

Data Storage and Distribution Solutions

Numerous protocols and applications partially address the objectives of this thesis. The primary focus of this work lies in the intersection of three domains: secure data storage, peer-to-peer data sharing, and distributed filesystems. To provide a comprehensive overview, this chapter examines existing implementations and solutions in each of these domains, analyzing their approaches, strengths, and limitations.

The remainder of this chapter is structured as follows:

- **Secure Data Storage:** This section discusses secure data storage, as implemented by the Linux Unified Key Setup (LUKS) system.
- **Peer-to-Peer Data Sharing:** Here, we delve into protocols and applications that enable decentralized sharing of data among peers, focusing on scalability, reliability, and performance.
- **Distributed Filesystems:** This section reviews distributed filesystem architectures, with an emphasis on their design principles, data consistency models, and fault tolerance.

Each of these sections evaluates relevant existing works with an emphesis on what they do not address. By synthesizing insights from these domains, the groundwork is laid for the development of a novel solution that integrates the best practices and innovative approaches from these fields.

1.1 Secure Data Storage

Protecting data confidentiality and integrity at rest is a fundamental requirement for many systems, particularly those involving sensitive information or operating in untrusted environments. Various mechanisms exist to achieve this, operating at different levels of the storage stack, from entire disks to individual files. This section reviews prominent approaches, analyzing their design, features, and inherent limitations. Special attention is paid to their suitability for secure data sharing, or lack thereof.

1.1.1 Disk and Partition Encryption

Full-Disk Encryption (FDE) and partition encryption aim to protect all data stored on a specific block device or partition. By encrypting the entire storage medium below the filesystem layer, these methods provide comprehensive protection against unauthorized physical access to the storage device [41].

Linux Unified Key Setup

LUKS is the standard for disk encryption within the Linux ecosystem [17]. It provides a platform-independent specification for managing encryption keys and metadata. LUKS itself does not perform the encryption; it acts as a wrapper around underlying kernel crypto modules (like dm-crypt). Key features include:

- **Metadata Header:** Contains the encryption cipher, mode, key size, and multiple key slots.
- **Key Slots:** Allows multiple passphrases or keys to decrypt the same master key, facilitating key management and revocation without re-encrypting the entire volume. LUKS2, the newer version, enhances flexibility and metadata resilience [17].
- **Detached Headers:** LUKS2 allows storing the header separately from the encrypted data, potentially enhancing security or enabling specific use cases.

While LUKS provides robust protection for data at rest on a specific device, it is inherently tied to that block device. It does not directly address the secure sharing of data between different systems or peers. Data must be decrypted before it can be accessed or shared, requiring separate mechanisms to secure the data in transit or on the recipient's system.

Proprietary full-disk encryption options

Other notable disk encryption systems include **BitLocker**, integrated into Microsoft Windows, and **FileVault**, used in Apple's macOS. These systems offer similar FDE capabilities, but are platform-specific, and often rely on hardware security modules like the Trusted Platform Module (TPM) for key protection. Their proprietary nature alone renders their consideration here unconscionable.

1.1.2 Filesystem-Level Encryption

Encryption can also be implemented at the filesystem level. Systems such as **eCryptfs** [21] and the native encryption features in filesystems like **ZFS** [9] and **ext4 (fscrypt)** [1] operate above the block layer but below the application layer. This enables finer control over encrypted data, such as assigning different keys to individual directories or files. However, these systems are designed for single-host protection and lack support for secure data exchange with external peers, including mechanisms for key distribution and integrity in distributed peer-to-peer networks.

1.1.3 File-Level and Container Encryption

File-level encryption targets individual files, with standards like **OpenPGP** [10] enabling encryption and signing via public-key cryptography, as seen in tools like GnuPG. This

approach is suitable for isolated transfers, such as secure email, but it struggles with scalable key management in dynamic peer-to-peer environments. Container-based tools like **VeraCrypt** [22] create encrypted containers that function as virtual volumes, providing portability for stored data. However, these containers require full-file transfers and lack built-in support for distributed key sharing and integrity verification, rendering them unsuitable for secure peer-to-peer data exchange.

1.1.4 Applicability to Peer-to-Peer Networks

Existing mechanisms for secure data storage have limited applicability in decentralized peer-to-peer networks, even though they are effective for protecting data at rest. The key factors limiting their use are:

- **Granularity and Efficiency:** These systems often encrypt entire volumes or files, which is inefficient for selective sharing of data blocks of arbitrary sizes, as required by a general peer-to-peer data sharing protocol.
- **Key Management:** They do not provide scalable key distribution among untrusted nodes, making them impractical for dynamic peer-to-peer environments.
- **Portability and Interoperability:** Platform-specific designs and the need for full-file transfers hinder smooth operation in heterogeneous peer-to-peer networks.
- **Access Control:** There is a lack of fine-tuned access settings, which would be necessary for scenarios where different nodes require varying levels of data access.

These limitations underscore the need for a specialized protocol optimized for data sharing in peer-to-peer networks.

1.2 Peer-to-Peer Data Sharing Systems

Peer-to-peer data sharing systems enable decentralized distribution of data among nodes without a central server. [36] These systems prioritize efficient content dissemination and scalability, but often fall short of meeting comprehensive security requirements for sensitive data. Specifically:

- **Confidentiality and Integrity:** They lack robust encryption for data in transit or at rest on intermediate peers, making them susceptible to interception and eavesdropping. While systems like **BitTorrent** use cryptographic hashes to verify the integrity of data pieces against torrent metadata [11], thereby preventing corruption or tampering, they do not provide end-to-end confidentiality for shared data.
- **Key Management and Access Control:** Secure key exchange mechanisms and fine-grained access controls are generally absent, complicating trust management among nodes and exposing data to unauthorized access or modification.

Consequently, these limitations underscore the need for a protocol specifically designed to provide robust confidentiality, authenticated integrity, flexible key management, and granular data control in peer-to-peer environments.

1.3 Distributed Filesystems

Distributed filesystems (DFS) aim to provide a unified view of data stored across multiple network nodes, often employing techniques like content addressing for data retrieval and deduplication. [43] A prominent example is the InterPlanetary File System (**IPFS**) [4], which uses content identifiers (CIDs) based on cryptographic hashes to address data immutably. While DFS architectures excel at data availability and censorship resistance, their design isn't suitable for secure, private data sharing among specific peers:

- **Confidentiality by Default:** Many DFS, including IPFS by default, focus on public data accessibility. While data can be encrypted before being added, the systems themselves do not typically mandate or seamlessly integrate end-to-end encryption and key management for private peer-to-peer sharing use cases. Accessing private data often requires separate, out-of-band key exchange mechanisms.
- **Granular Access Control:** Implementing fine-grained access control (e.g., granting specific permissions to different users or groups for subsets of data) can be complex or non-native in content-addressed systems designed primarily for immutable data retrieval based on knowing the hash.
- **Key Management Complexity:** Securely managing and distributing cryptographic keys for encrypted data shared among dynamic groups of peers remains a significant challenge that is often outside the core scope of the DFS protocol itself.
- **Metadata Privacy:** Information about data access patterns or even the existence of certain data (via CIDs) might be exposed to network participants, which can be undesirable for sensitive information.

Therefore, while distributed filesystems offer powerful primitives for decentralized data storage and retrieval, their standard implementations often lack the integrated confidentiality, access control, and key management features required for a generalized, secure peer-to-peer data sharing protocol.

Chapter 2

Symmetric Encryption Algorithms and Authenticated Encryption

This chapter delves into the domain of symmetric cryptography, a cornerstone for ensuring data security in modern networked systems. It begins by establishing the core principles of symmetric encryption, differentiating between key cipher types, and underscoring the critical need for authenticated encryption to achieve comprehensive data protection. Subsequently, the chapter undertakes a detailed review of selected prominent algorithms and authenticated encryption schemes, culminating in a comparative analysis of their performance and security attributes. The overarching aim is to provide a foundational understanding of these cryptographic primitives, which are essential for the design and implementation of secure data sharing protocols.

2.0.1 Core Principles of Symmetric Encryption

Symmetric encryption, a foundational paradigm in modern cryptography, employs a single secret key shared between communicating entities for both the encryption of plaintext into ciphertext and the subsequent decryption of ciphertext back into plaintext [40, 26]. The principal objective of symmetric encryption is the provision of data confidentiality, thereby precluding unauthorized access to sensitive information. This chapter presents a comparative analysis of symmetric encryption algorithms, evaluating their performance and security characteristics. The study commences with an examination of block ciphers, a significant and historically foundational category within symmetric encryption.

2.0.2 Symmetric Cipher Types: Block and Stream Ciphers

Symmetric ciphers are primarily classified based on their method of processing plaintext data into two principal types: block ciphers and stream ciphers [42, 37]. Block ciphers operate on fixed-size groups of bits, termed blocks (e.g., 128 bits), transforming an entire plaintext block into a ciphertext block of identical size using the secret key. Since messages often exceed the length of a single block, **modes of operation** are defined to specify how a block cipher's core encryption function should be repeatedly and securely applied to encrypt sequences of blocks, often incorporating techniques to ensure that identical plaintext blocks do not produce identical ciphertext blocks. Conversely, stream ciphers process plaintext in smaller, sequential units, typically bits or bytes. This is achieved by generating a pseudorandom keystream from the secret key and a unique

nonce (a number used once) or an initialization vector (IV); this keystream is then combined (commonly via an XOR operation) with the plaintext units. This fundamental distinction in operational methodology results in differing performance profiles and distinct areas of applicability for each cipher type.

2.0.3 The Imperative of Authenticated Encryption

While confidentiality, achieved through symmetric encryption, constitutes a primary security objective, it is often insufficient in isolation to guarantee the comprehensive security of data, whether in transit or at rest. Security properties of commensurate criticality include data integrity, which ensures that data has not been illicitly altered, and data authenticity, which verifies the data's purported origin [16]. In the absence of robust integrity and authenticity mechanisms, an adversary could potentially modify encrypted messages without detection or inject maliciously crafted messages that appear legitimate, thereby undermining the overall security goals.

Message Authentication Codes (MACs) are cryptographic primitives specifically engineered to provide these vital assurances of data integrity and authenticity. [40] A MAC algorithm, utilizing a shared secret key, processes a message to generate a short, fixed-length tag. This tag enables a recipient who possesses the same secret key to verify that the message has not been tampered with during transmission and that it originates from a party privy to the key.

Although encryption and MACs can be applied as distinct cryptographic operations, the secure composition of these primitives, for instance, through generic approaches such as encrypt-then-MAC, demands meticulous design and analysis to preclude subtle yet critical vulnerabilities. Indeed, naive or improper combinations can lead to insecure schemes despite the strength of the individual components [3]. To address these composition challenges and provide a more robust, integrated cryptographic solution, Authenticated Encryption with Associated Data (AEAD) schemes have been developed. AEAD schemes are designed to holistically and securely integrate an encryption algorithm with an authentication mechanism (typically MAC-based). This integration is engineered to concurrently provide confidentiality, integrity, and authenticity for the encrypted data, as well as integrity and authenticity for any associated unencrypted data, thereby offering protection against a broader spectrum of sophisticated attacks [15].

2.0.4 Scope of Algorithms Under Review

This chapter presents a detailed review of selected contemporary and secure symmetric cryptographic primitives, focusing on options that represent realistic candidates for modern secure protocols. The examination begins with prominent block ciphers: the Advanced Encryption Standard (AES), the prevailing global standard for symmetric encryption [33], will be analyzed alongside Serpent, an algorithm recognized for its high security margin and conservative design principles stemming from the AES competition [2]. Subsequently, the discussion will address ChaCha20, a high-performance stream cipher specifically optimized for efficient software implementations across a variety of CPU architectures [7]. Recognizing the paramount importance of ensuring both confidentiality and integrity, the analysis will culminate with an investigation of leading Authenticated Encryption with Associated Data (AEAD) schemes. Specifically, AES-GCM (Galois/Counter Mode), which combines AES in Counter Mode with the GMAC authenticator [15], and ChaCha20-Poly1305, which pairs the ChaCha20 stream cipher with the Poly1305 message authenticator [35], will be ex-

amined. The selection of these particular algorithms and AEAD schemes is predicated on their status as robust, contemporary cryptographic solutions, their well-established security credentials, and their collective utility in establishing a comprehensive comparative basis for evaluating performance and security characteristics relevant to current system design.

2.1 Foundational Symmetric Ciphers

This section provides a detailed examination of the core symmetric encryption algorithms selected for review: the Advanced Encryption Standard (AES), Serpent, and ChaCha20. Each algorithm is analyzed with respect to its historical context, underlying design principles, and fundamental operational characteristics, thereby establishing a basis for subsequent comparative analysis.

2.1.1 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) represents a cornerstone of contemporary symmetric cryptography, having achieved widespread global adoption due to its robust security and efficient performance. Its development through a public, competitive process set a precedent for cryptographic standardization.

Algorithm Overview and Standardization

The genesis of AES lies in the U.S. National Institute of Standards and Technology (NIST) initiative, formally announced in 1997, to identify a successor to the Data Encryption Standard (DES), which was becoming increasingly vulnerable due to its relatively small key size [32]. This initiative invited submissions of candidate encryption algorithms from the international cryptographic community. The evaluation criteria emphasized not only security against known attacks but also computational efficiency, memory footprint, and adaptability across diverse hardware and software platforms [32]. Following an intensive multi-year public review and analysis period, the Rijndael [12] algorithm, developed by Belgian cryptographers Joan Daemen and Vincent Rijmen, was announced as the selected candidate in October 2000 [33]. Subsequently, in November 2001, NIST formally adopted Rijndael as the Advanced Encryption Standard, codifying it in Federal Information Processing Standards Publication (FIPS PUB) 197 [33]. AES has since been globally embraced, forming an integral component of numerous international standards and a vast array of secure communication protocols and commercial products.

Key Operational Characteristics

AES is a symmetric block cipher adhering to the Substitution-Permutation Network (SPN) design paradigm. It operates on a fixed data block size of 128 bits and supports three distinct cryptographic key lengths: 128 bits, 192 bits, and 256 bits [33]. The encryption process involves a series of iterative rounds, the number of which is determined by the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys [13].

Each round, with the exception of the final round which has a slightly modified structure, comprises four distinct byte-oriented transformations applied to an internal state, typically represented as a 4×4 matrix of bytes:

- **SubBytes:** A non-linear byte substitution, where each byte of the state is independently replaced with another byte according to a fixed substitution table, known as the S-box. This transformation introduces confusion into the ciphertext.
- **ShiftRows:** A linear transposition step, where the bytes in the last three rows of the state matrix are cyclically shifted by different offsets. This operation provides diffusion by spreading byte influences across columns.
- **MixColumns:** A linear mixing operation that processes each column of the state independently. The four bytes of each column are treated as coefficients of a polynomial over the finite field $GF(2^8)$ and multiplied by a fixed polynomial. This transformation, along with ShiftRows, ensures strong diffusion over multiple rounds. (This step is omitted in the final round).
- **AddRoundKey:** Each byte of the state is combined with a corresponding byte of a round key using a bitwise XOR operation. The round keys are derived from the primary cipher key through a key expansion algorithm (key schedule).

The structured, iterative application of these transformations ensures that the resultant ciphertext is a highly complex, non-linear function of both the plaintext and the secret key, thereby providing robust security against a wide range of cryptanalytic techniques, including linear and differential cryptanalysis, when correctly implemented [13, 33].

2.1.2 Serpent

Serpent is a symmetric key block cipher, notable for being one of the five finalists in the Advanced Encryption Standard (AES) competition. It was designed by Ross Anderson, Eli Biham, and Lars Knudsen [2]. Serpent's design philosophy prioritized a high security margin and a conservative approach, aiming for straightforward security analysis and robustness against then-known and anticipated cryptanalytic techniques.

Algorithm Overview and Design Rationale

The designers of Serpent explicitly aimed for a cipher that would be exceptionally secure, even if this came at some cost to performance compared to other AES candidates [2]. The core design principle was to use well-understood cryptographic components and a structure that allowed for a relatively clear security analysis. Serpent operates on 128-bit blocks of data and supports key sizes of 128, 192, or 256 bits, similar to AES. The algorithm consists of 32 rounds, a significantly higher number than many contemporary block ciphers, which contributes to its high security margin. Each round applies the same sequence of operations, simplifying the design and analysis. A key feature of Serpent is its use of a series of eight distinct 4-bit by 4-bit substitution boxes (S-boxes) per round. These S-boxes were chosen based on their strong non-linearity and resistance to differential and linear cryptanalysis. The operations within each round were designed to be efficiently implementable using bit-slice techniques, allowing for parallel computation on modern processors, which can mitigate some of the performance overhead from the high round count [2].

Key Operational Characteristics

Serpent's structure is a Substitution-Permutation Network (SPN), similar to AES. The 32 rounds are a defining characteristic. In each round, the following operations are performed:

- **Key Mixing:** The 128-bit round key (derived from the main key via a key schedule) is XORed with the current state.
- **S-box Application:** The 128-bit state is passed through a layer of S-boxes. Specifically, the state is viewed as thirty-two 4-bit words, and each 4-bit word is substituted using one of the eight S-boxes. The S-box used in a particular position changes from round to round in a predefined sequence.
- **Linear Transformation:** After the S-box layer, a linear transformation is applied to the entire 128-bit state. This transformation provides diffusion, ensuring that the output of one S-box in a round affects the input of multiple S-boxes in the next round. This transformation is fixed across all rounds.

An initial permutation (IP) is applied to the plaintext before the first round, and a final permutation (FP) is applied after the 32nd round to produce the ciphertext. These permutations are inverses of each other and are designed to simplify optimized implementations rather than contributing directly to the cryptographic strength against standard attacks [2]. While Serpent's high number of rounds and conservative S-box design contribute to its strong security reputation, they also generally result in lower software performance compared to AES, particularly when dedicated hardware acceleration for AES is available.

2.1.3 ChaCha20

ChaCha20 is a stream cipher developed by Daniel J. Bernstein as a refinement of his earlier Salsa20 algorithm [7]. Unlike block ciphers that encrypt fixed-size blocks of data, stream ciphers generate a pseudorandom keystream which is then combined (typically via XOR) with the plaintext to produce ciphertext. ChaCha20 has gained significant traction due to its strong security profile, excellent performance in software implementations, and simple design.

Algorithm Overview and Design Rationale

The primary design goal for ChaCha20 was to provide high security with exceptional speed in software, particularly on general-purpose CPUs that may lack dedicated hardware support for cryptographic operations [7]. It aims to offer better diffusion per round compared to Salsa20, thereby potentially achieving similar or greater security with fewer rounds, although the standard ChaCha20 specification uses 20 rounds (or 8 or 12 for reduced-round variants). The algorithm is designed to be resistant to timing attacks and other software side-channel vulnerabilities due to its consistent, data-independent operational flow. ChaCha20 operates with a 256-bit key and requires a 96-bit nonce (number used once) and a 32-bit block counter for each invocation to generate a unique keystream block. The uniqueness of the nonce is critical for its security [35].

Key Operational Characteristics

ChaCha20 generates 512-bit (64-byte) blocks of keystream by iteratively applying a core function to an internal state matrix. This state is initialized with constants, the 256-bit key, the 32-bit counter, and the 96-bit nonce [35]. The internal state is a 4×4 matrix of 32-bit words. The core of the ChaCha20 algorithm is the „quarter-round“ operation, which is applied repeatedly to the columns and diagonals of the state matrix. A quarter-round takes four 32-bit words (a, b, c, d) and updates them as follows:

1. $a \leftarrow a \ + b$; $d \leftarrow d \oplus a$; $d \leftarrow (d \lll 16)$
2. $c \leftarrow c + d$; $b \leftarrow b \oplus c$; $b \leftarrow (b \lll 12)$
3. $a \leftarrow a \ + b$; $d \leftarrow d \oplus a$; $d \leftarrow (d \lll 8)$
4. $c \leftarrow c + d$; $b \leftarrow b \oplus c$; $b \leftarrow (b \lll 7)$

Where $+$ denotes addition modulo 2^{32}, $\oplus$ denotes bitwise XOR, and $\lll n$ denotes a left rotation by n bits. A full ChaCha20 round consists of applying the quarter-round operation to each of the four columns and then to each of the four diagonals (sometimes referred to as „column rounds“ and „diagonal rounds“). The standard ChaCha20 algorithm performs 20 such rounds (or 10 iterations of column rounds and diagonal rounds). After the rounds are completed, the resulting state matrix is added (word-wise) to the initial state matrix to produce the 512-bit keystream block. This keystream block is then XORed with the corresponding block of plaintext to produce ciphertext, or with ciphertext to produce plaintext. For subsequent blocks of a message, the block counter is incremented, and the process is repeated to generate the next keystream block [35, 7].

2.2 Authenticated Encryption with Associated Data (AEAD) Schemes

The provision of data confidentiality via symmetric encryption, as discussed in previous sections, is a primary security service. However, in many practical applications, confidentiality alone is insufficient; assurances of data integrity (that data has not been modified) and data authenticity (that data originates from the purported source) are equally critical. Authenticated Encryption with Associated Data (AEAD) schemes are cryptographic constructions designed to simultaneously provide these three security services: confidentiality, integrity, and authenticity, for a given plaintext, and additionally to provide integrity and authenticity for any associated data that is transmitted in the clear alongside the ciphertext [15].

2.2.1 Principles and Advantages of AEAD

Authenticated Encryption (AE) refers to a symmetric encryption scheme that transforms a plaintext into a ciphertext such that the scheme provides both confidentiality for the plaintext and a proof of its integrity and authenticity [39]. The extension to AEAD allows for the inclusion of „associated data“ (AD), also termed „additional authenticated data“ (AAD). This AD is not encrypted but is protected against modification; common examples include protocol headers or other contextual metadata that must remain unencrypted but whose integrity is vital for the correct and secure processing of the encrypted payload [15].

The development and adoption of AEAD schemes stem from several key advantages over ad-hoc compositions of separate encryption and Message Authentication Code (MAC) algorithms:

- **Integrated Security Properties:** AEAD schemes are designed as monolithic primitives that bind confidentiality with integrity and authenticity. This integrated design aims to prevent vulnerabilities that can arise from the improper or insecure composition of distinct encryption and MAC functions. While generic composition paradigms (e.g., Encrypt-then-MAC) can achieve security, their correct implementation is subtle and error-prone, with certain combinations known to be insecure [3, 28]. Dedicated AEAD constructions are often accompanied by security proofs under standard cryptographic assumptions.
- **Simplified Cryptographic Interface:** By providing a single interface for encryption and authentication (and their respective inverse operations), AEAD schemes reduce the complexity for application developers. This simplification minimizes the potential for errors in cryptographic protocol design and implementation, such as incorrect sequencing of operations or mishandling of intermediate values, which can otherwise lead to security failures [18].
- **Resistance to Chosen-Ciphertext Attacks:** Many well-established AEAD schemes are designed to be secure against adaptive chosen-ciphertext attacks (IND-CCA2). This security notion implies that an adversary, even one capable of obtaining decryptions of arbitrarily chosen ciphertexts (except the target ciphertext itself), cannot gain information about the plaintext corresponding to a target ciphertext, nor can they forge a valid ciphertext for a new message [26].
- **Efficiency Considerations:** Modern AEAD schemes are often engineered for high performance. Algorithms such as AES-GCM can leverage hardware acceleration for the underlying block cipher, while others like ChaCha20-Poly1305 are optimized for fast software execution. In many instances, the computational overhead of the integrated authentication is minimal compared to the encryption operation itself [15, 35].

Consequently, the use of AEAD schemes is widely recognized as a best practice in cryptographic engineering for applications requiring robust protection of symmetric-key encrypted data [16].

2.2.2 AES-GCM (Galois/Counter Mode)

AES-GCM is a widely adopted AEAD scheme standardized by NIST in Special Publication 800-38D [15]. It combines the AES block cipher operating in Counter (CTR) mode for encryption with the Galois Message Authentication Code (GMAC) for authentication.

Construction and Operation

In AES-GCM, confidentiality is provided by AES-CTR mode. A unique nonce is used to generate a sequence of counter blocks, which are then encrypted with AES to produce a keystream. This keystream is XORed with the plaintext to produce the ciphertext. The same nonce and key must not be reused for different messages. Authentication is provided by GMAC. Both the ciphertext and any associated data (AAD) are processed by a universal hash function (GHASH), which is based on multiplication in the Galois Field

$GF(2^{128})$. The result of this hashing process is then encrypted with AES (using a key derived from the main encryption key and the nonce, typically by encrypting a block of all zeros) to produce the authentication tag [31]. The GHASH function requires a secret hash key H, which is derived by encrypting a 128-bit block of zeros with the AES key. The key operational steps involve:

1. Encryption of the plaintext using AES-CTR mode.
2. Computation of the GMAC tag over the AAD and the ciphertext.

During decryption, the ciphertext is decrypted using AES-CTR, and the GMAC tag is recomputed using the received AAD, the decrypted plaintext (or the received ciphertext, depending on the specific GCM variant/implementation detail), and the same key and nonce. If the computed tag matches the received tag, the decryption is considered valid, and the plaintext is released; otherwise, both the plaintext and the tag are rejected, indicating a potential integrity or authenticity failure [15].

Security and Performance Aspects

AES-GCM offers strong, provable security guarantees when the nonce is unique for every encryption with a given key [15]. Nonce reuse can lead to catastrophic failures, potentially compromising both confidentiality and authenticity. The security of GCM relies on the security of the underlying AES block cipher and the properties of the GHASH function. Performance-wise, AES-GCM can be very efficient, especially on platforms with hardware support for AES (e.g., AES-NI instruction sets) and carry-less multiplication (for GHASH, e.g., PCLMULQDQ instruction). Its parallelizable nature for both encryption (CTR mode) and authentication (GHASH processing) contributes to high throughput [20].

2.2.3 ChaCha20-Poly1305

ChaCha20-Poly1305 is another prominent AEAD scheme, standardized for IETF protocols in RFC 8439 [35]. It combines the ChaCha20 stream cipher for encryption with the Poly1305 message authenticator.

Construction and Operation

In this construction, the ChaCha20 stream cipher is used to encrypt the plaintext. A unique nonce and a block counter are used with the 256-bit key to generate the keystream, which is XORed with the plaintext. The Poly1305 authenticator is then used to compute a 128-bit tag over the associated data and the ciphertext. Poly1305 requires a 256-bit one-time key: 128 bits for the Poly1305 function itself (often denoted as 'r' and 's', with clamping applied to 'r') and a 128-bit nonce for the final AES encryption in some older Poly1305-AES variants, or more commonly in the ChaCha20-Poly1305 context, a one-time key derived from the main key and nonce specifically for Poly1305. For the IETF ChaCha20-Poly1305 AEAD, a special Poly1305 key is generated by encrypting a block of zeros (or the first block of the ChaCha20 keystream, specifically the first 32 bytes from the ChaCha20 block function with block counter 0) using the ChaCha20 key and nonce [35]. The general procedure is:

1. Generate the Poly1305 one-time key using ChaCha20 (with block counter 0).

2. Encrypt the plaintext using ChaCha20 (with block counter starting from 1).

3. Compute the Poly1305 tag over the AAD, the ciphertext, and padding, followed by the lengths of AAD and ciphertext.

During decryption, the ciphertext is decrypted, and the Poly1305 tag is recomputed. If the tags match, the plaintext is considered authentic and confidential [35].

Security and Performance Aspects

ChaCha20-Poly1305 is designed for high security and performance, particularly in software implementations where hardware AES acceleration may not be available [7, 6]. The security relies on the underlying strength of ChaCha20 as a stream cipher and Poly1305 as a secure MAC. Similar to AES-GCM, the uniqueness of the nonce per key is critical for security. Nonce reuse can compromise both confidentiality and authenticity. ChaCha20 is fast in software due to its simple arithmetic operations (additions, rotations, XORs) that map well to general-purpose CPU instructions. Poly1305 is also exceptionally fast in software. Consequently, the combined AEAD scheme offers excellent performance on a wide range of devices, from high-end servers to resource-constrained mobile and embedded systems [29].

2.3 Comparative Analysis: Performance and Security

The selection of an appropriate symmetric encryption algorithm or an Authenticated Encryption with Associated Data (AEAD) scheme for a given application necessitates a comparative evaluation of candidate primitives based on established research and performance data. This section synthesizes existing knowledge to compare the previously discussed algorithms, specifically AES, Serpent, and ChaCha20, and the AEAD schemes AES-GCM and ChaCha20-Poly1305. The analysis focuses on their documented performance characteristics in various environments, their established security profiles, and relevant implementation considerations, drawing upon published cryptographic studies and benchmark results.

2.3.1 Performance

Computational performance is a critical factor in the practical deployment of cryptographic algorithms. Key aspects include software implementation speed, the impact of hardware acceleration, latency, key setup times, and resource utilization.

Software Implementation Speed

In software implementations, particularly on platforms lacking dedicated cryptographic hardware, the architectural design of a cipher significantly influences its speed. ChaCha20 was specifically designed for high performance in software across a wide range of general-purpose CPUs. Benchmarking results, such as those from the eBACS (ECRYPT Benchmarking of Cryptographic Systems) project, consistently show ChaCha20 achieving excellent throughput, often outperforming software implementations of AES when AES-NI hardware acceleration is not utilized [8]. For instance, on many common x86-64 architectures, ChaCha20 can process data at several cycles per byte, making it highly competitive [7].

AES, while capable of high speeds, sees its software performance vary more significantly depending on the implementation quality and CPU architecture. Without hardware support, its architectural reliance on S-box lookups and the MixColumns operation can lead to relatively slower software implementations compared to the arithmetic-centric design of ChaCha20, a performance difference often observed in benchmarks [13, 8]. Serpent, with its 32 rounds and conservative design, generally exhibits lower software throughput compared to both AES and ChaCha20, as its primary optimization was for security margin rather than raw speed [2].

When comparing the AEAD schemes, ChaCha20-Poly1305 inherits the strong software performance of its constituent parts. The Poly1305 authenticator is exceptionally fast in software [6]. Consequently, ChaCha20-Poly1305 often demonstrates superior throughput to AES-GCM in software-only environments, especially on platforms where AES hardware acceleration is absent or on architectures (e.g., some ARM processors) where ChaCha20's operations are particularly well-suited [35, 8]. AES-GCM's software performance is tied to the software speed of AES and the GHASH computation; the latter can be a bottleneck without specific hardware support for carry-less multiplication.

Impact of Hardware Acceleration

The performance landscape of symmetric ciphers is dramatically altered by the availability of dedicated hardware acceleration features in modern processors, particularly for block ciphers like AES. The Advanced Encryption Standard New Instructions (AES-NI) are a set of extensions to the x86 instruction set architecture, introduced by Intel and subsequently adopted by AMD, specifically designed to accelerate AES encryption, decryption, and key generation operations [19]. The utilization of AES-NI can result in a substantial increase in AES throughput, often by an order of magnitude or more compared to purely software-based implementations, thereby rendering AES exceptionally efficient on supporting hardware platforms [19]. This acceleration extends to AEAD schemes based on AES, such as AES-GCM. For AES-GCM, beyond AES-NI for the encryption component, the PCLMULQDQ instruction (Carry-Less Multiplication) provides hardware support for accelerating the GHASH polynomial multiplication in $GF(2^{128})$, which is critical for the performance of the authentication component [20].

In contrast, stream ciphers like ChaCha20, and consequently the AEAD scheme ChaCha20-Poly1305, were primarily designed to achieve high performance in software without reliance on specialized hardware instructions [7]. This design philosophy ensures consistent and strong performance across a broad spectrum of CPU architectures, including those that lack dedicated cryptographic accelerators (e.g., many ARM-based processors found in mobile and embedded devices, or older x86 systems). While ChaCha20 does not benefit from instruction sets tailored for AES, its arithmetic operations (additions, rotations, XORs) are generally well-suited to modern CPU pipelines. As a result, in environments where AES-NI [19] and PCLMULQDQ [20] are available and effectively leveraged by the cryptographic library, AES-GCM typically exhibits superior raw throughput. However, on platforms devoid of such hardware support, ChaCha20-Poly1305 often demonstrates a significant performance advantage due to its software-optimized nature [8]. The relative performance is therefore highly contingent on the target hardware capabilities.

Latency, Key Setup, and Resource Usage

Beyond aggregate throughput, several other performance-related factors influence the suitability of cryptographic algorithms for diverse application scenarios. Latency, which quantifies the delay in processing a minimal data unit (e.g., a single block for block ciphers or an initial data segment for stream ciphers), is of paramount importance for interactive and real-time communication systems. Block ciphers such as AES and Serpent, when encrypting a single block, exhibit latency characteristics primarily dictated by their respective round counts and the computational complexity of their round functions. Stream ciphers, exemplified by ChaCha20, can offer exceptionally low latency for initiating data processing once the keystream generation is underway, as the keystream can be XORed with plaintext bytes immediately upon their availability [7]. For AEAD schemes, the overall latency encompasses the time taken for both the encryption/decryption phase and the generation or verification of the authentication tag.

The key setup time, representing the computational effort required to initialize a cipher with a new cryptographic key (and, where applicable, a nonce or initialization vector), can become a significant performance consideration in contexts involving frequent re-keying or the processing of numerous small, independently keyed messages. The AES algorithm incorporates a key schedule that can be relatively complex, particularly for 256-bit keys, potentially contributing to a non-negligible setup time in certain implementations [13]. Serpent's key schedule, while different, also involves a preparatory phase. In contrast, ChaCha20 is distinguished by a notably simple and rapid key setup procedure, which largely consists of arranging the key and nonce directly into its initial state matrix, a characteristic that can be advantageous in environments with high session turnover or frequent context switches [35].

Finally, resource utilization, which includes the code size of the implementation and its operational memory footprint (e.g., for storing S-boxes, expanded round keys, or the internal cipher state), is a critical determinant for deployment in resource-constrained environments, such as embedded systems, Internet of Things (IoT) devices, or secure elements. AES implementations, especially those that precompute S-box lookup tables and round keys for performance, can have a moderate memory requirement. Serpent's S-boxes can either be generated algorithmically during its key schedule or precomputed, which affects its memory demands [2]. ChaCha20, by virtue of its design which eschews large S-boxes and maintains a straightforward internal state, is generally characterized by a compact code size and minimal RAM requirements, rendering it highly suitable for such constrained platforms [7]. The overall resource demands of AEAD schemes like AES-GCM and ChaCha20-Poly1305 are predominantly influenced by the characteristics of their underlying cipher and MAC components.

2.3.2 Security Profile and Considerations

Beyond computational performance, the paramount criterion for selecting cryptographic primitives is their security. This involves an assessment of their theoretical strength against known cryptanalytic attacks, the practical security offered by specific modes of operation or constructions, and considerations related to their secure implementation and usage.

Core Cipher Strength

The foundational security of an AEAD scheme is intrinsically linked to the cryptographic strength of its underlying core cipher (block or stream cipher). The Advanced Encryption Standard (AES) has undergone extensive public scrutiny for over two decades and is considered secure against all known practical cryptanalytic attacks when used with appropriate key lengths (128, 192, or 256 bits) [33, 13]. Its security is founded upon the extensively analyzed properties of Substitution-Permutation Networks and the demonstrable resistance of its round structure to cryptanalysis.

Serpent, an AES finalist, was designed with an even more conservative approach, featuring 32 rounds and S-boxes with strong mathematical properties intended to provide a very high security margin against techniques like linear and differential cryptanalysis [2]. While this conservative design generally leads to lower performance than AES, Serpent is widely regarded as having a very strong theoretical security profile.

ChaCha20, a stream cipher, derives its security from the difficulty of distinguishing its output keystream from a truly random stream, given the key and nonce. It is based on the well-analyzed ARX (Add-Rotate-XOR) design principles. Extensive cryptanalysis has not revealed practical weaknesses in the full 20-round ChaCha20, and it is considered secure for its intended applications [7]. Its security is critically dependent on the uniqueness of the nonce for each key.

AEAD Scheme Security

The security of an Authenticated Encryption with Associated Data (AEAD) scheme depends not only on the strength of its underlying cipher and authenticator but also critically on the secure composition of these components and the proper handling of operational parameters such as nonces.

AES-GCM (Galois/Counter Mode) provides strong security guarantees, achieving confidentiality through AES in Counter (CTR) mode and integrity/authenticity via the GMAC algorithm [15]. Its security is proven under the assumption that AES is a secure pseudorandom permutation (PRP) and that the GHASH function is a secure universal hash function [31]. However, the security of AES-GCM is exceptionally sensitive to nonce reuse. If a nonce is ever repeated with the same key for two different messages, an attacker can compromise the confidentiality of both messages (due to the properties of CTR mode) and potentially forge authentication tags [39, 15]. While AES-GCM is highly efficient, especially with hardware support, this strict nonce uniqueness requirement necessitates careful implementation and nonce management strategies.

ChaCha20-Poly1305 offers robust authenticated encryption by combining the ChaCha20 stream cipher with the Poly1305 message authenticator [35]. Its security relies on the pseudorandomness of ChaCha20 and the unforgeability of Poly1305. Similar to AES-GCM, the security of ChaCha20-Poly1305 is critically dependent on the uniqueness of the nonce for each key; nonce reuse can lead to a loss of confidentiality and allow tag forgery [35]. From a side-channel resistance perspective, ChaCha20, being an ARX cipher with a data-independent operational flow, can offer advantages in software implementations compared to table-lookup-based ciphers like AES if not protected by constant-time implementations or hardware support [7]. Poly1305 is also designed with consideration for side-channel resistance [6]. The overall construction of ChaCha20-Poly1305 is considered state-of-the-art, providing strong protection when nonces are handled correctly.

Both AES-GCM and ChaCha20-Poly1305 are designed to protect against common attacks on combined encryption and authentication, such as those exploiting weaknesses in generic composition methods [3]. The choice between them often involves a trade-off between hardware acceleration availability (favoring AES-GCM) and consistent software performance with potentially better inherent side-channel resistance in software (favoring ChaCha20-Poly1305), alongside considerations for nonce generation capabilities within the target system.

Implementation

The theoretical security of a cryptographic algorithm provides a foundational assurance, yet its practical security in deployed systems is profoundly influenced by the nuances of its implementation and the maturity of its surrounding ecosystem. Complex algorithms with strict operational requirements, such as stringent nonce management, can inadvertently increase the risk of implementation errors. Such errors, even if minor, can introduce critical vulnerabilities, thereby undermining the algorithm's intended security guarantees, regardless of its theoretical robustness [18].

The Advanced Encryption Standard (AES), benefiting from its status as a global standard, is supported by an extensive ecosystem. A vast array of optimized and rigorously vetted implementations is available across nearly all programming languages and hardware platforms, with some even offering direct hardware acceleration [33]. Nevertheless, achieving a secure AES implementation, particularly for its advanced modes of operation like GCM, demands meticulous attention to detail. This includes robust nonce generation strategies, and, in the absence of comprehensive hardware defenses, careful consideration of side-channel attack vectors [15]. Serpent, while recognized for its formidable theoretical security, is supported by a comparatively smaller ecosystem. Consequently, readily available, highly optimized libraries implementing it are less common, and its intricate structure, characterized by 32 rounds and multiple distinct S-boxes, may present a greater challenge for developers aiming for both efficient and verifiably secure custom implementations [2].

In contrast, ChaCha20 and Poly1305 were conceived with simplicity and the facilitation of secure software implementation as primary design objectives [7, 6]. Their reliance on basic arithmetic operations, which obviates the need for table lookups, can simplify the development of constant-time code, which is a critical technique for mitigating timing-based side-channel attacks [27]. The AEAD construction ChaCha20-Poly1305, standardized in RFC 8439, defines clear operational guidelines. High-quality implementations of this scheme are increasingly prevalent in modern cryptographic libraries, especially those tailored for internet security protocols and mobile computing environments [35]. The inherent straightforwardness of its design, coupled with its strong software performance, fosters a continually expanding and robust ecosystem.

In conclusion, the maturity, widespread availability, and trustworthiness of meticulously tested library implementations are of paramount importance. Relying upon such established libraries is strongly recommended in lieu of attempting bespoke cryptographic implementations. This preference is well-founded, as library maintainers typically possess specialized expertise in cryptographic engineering, secure coding practices, and side-channel analysis [16]. Consequently, the selection of a cryptographic algorithm is often pragmatically governed by the quality, accessibility, and proven reliability of its implementations within the specific development and deployment context.

2.4 Conclusion

This chapter has presented a comparative analysis of selected symmetric encryption algorithms and AEAD schemes. The review encompassed the Advanced Encryption Standard (AES), Serpent, the stream cipher ChaCha20, and the AEAD constructions AES-GCM and ChaCha20-Poly1305, evaluating their respective performance characteristics, security profiles, and pertinent implementation aspects.

The analysis underscored the distinct characteristics and inherent trade-offs associated with each cryptographic primitive. AES, as the prevailing global standard, offers a robust combination of security and efficiency, particularly when its performance is helped by dedicated hardware acceleration such as AES-NI. Its extensive and mature ecosystem provides considerable assurance. Serpent, while theoretically formidable due to its highly conservative design, typically incurs a notable software performance penalty when compared to AES. ChaCha20, engineered for optimal software execution, excels in environments lacking specialized cryptographic hardware, offering strong security coupled with a simple design that is well-suited for a diverse array of platforms.

In the domain of authenticated encryption, both AES-GCM and ChaCha20-Poly1305 are designed to deliver comprehensive security by synergistically integrating confidentiality with data integrity and authenticity. AES-GCM's performance is particularly compelling with appropriate hardware support, though its security critically depends on rigorous nonce management. ChaCha20-Poly1305 provides excellent software performance and benefits from its simple design, while similarly mandating strict nonce discipline. It must be emphasized that the practical, achieved security of any chosen scheme is invariably linked to its correct and secure implementation, underscoring the critical importance of utilizing mature, well-vetted cryptographic libraries.

Ultimately, this comparative study underscores the principle that no single symmetric cipher or AEAD scheme is universally superior across all possible applications. The optimal selection of cryptographic primitives is intrinsically context-dependent, demanding a meticulous evaluation of specific system requirements. Critical considerations include the target operational environment (including the availability of hardware acceleration), explicit performance expectations (regarding throughput, latency, and resource utilization), the desired security posture against relevant threat models (encompassing resistance to side-channel attacks), and the practical considerations for secure implementation and integration. The understanding gleaned from this analysis forms an indispensable foundation for making judicious cryptographic choices in the subsequent design of secure communication and data protection protocols.

Chapter 3

Use Cases and Key Requirements for Secure Data Sharing

The proliferation of decentralized technologies and the increasing need for privacy-preserving communication present a need for robust mechanisms to facilitate secure data sharing directly between peers. In contrast to centralized architectures, which inherently depend on trusted intermediaries, peer-to-peer systems present potential advantages concerning resilience, censorship resistance, and user autonomy. [36] Nevertheless, the realization of these benefits is frequently accompanied by significant security challenges, particularly when the data being shared is confidential or of a sensitive nature. This chapter undertakes an analysis of illustrative use cases for secure peer-to-peer data sharing to elucidate their diverse needs and underlying motivations. Subsequently, based upon this analysis, a set of fundamental requirements will be systematically derived. Any protocol engineered for secure peer-to-peer data sharing must address these requirements to be deemed effective and trustworthy; they will therefore inform the foundational design considerations for such a system.

3.1 Use Cases

To delineate the foundational requirements of a secure peer-to-peer data sharing system, it is helpful to examine several illustrative practical scenarios.[1] While not exhaustive, the following examples are presented to highlight common contexts in which such a system offers distinct advantages. This examination will subsequently inform the derivation of key system prerequisites.

3.1.1 Secure Collaboration for Dispersed Teams

A common scenario involves teams whose members are geographically distributed yet need to collaborate on projects involving highly sensitive information. For instance, consider **multi-institutional healthcare research consortia** analyzing anonymized but still sensitive patient data for medical advancements, or **legal teams from different offices** jointly preparing a complex case involving confidential client information and privileged documents. Such teams require a private and secure method for sharing data, managing

[1] The general use cases described form the motivations for this thesis. The specific examples mentioned in subsection 1 are hypothetical.

document versions, and ensuring that only authorized individuals can access the shared information. Crucially, this method should ideally avoid reliance on centralized third-party services, which can present risks such as data breaches, service outages, or external control over the data.

3.1.2 Private Content Distribution

Individuals or creators often need to share digital content, such as exclusive artistic works, journalistic information, or personal photographs and videos, directly with a specific audience, but may not wish to use public platforms. For these users, the primary objective is to ensure that only the intended recipients can access the content and that the sharing process itself maintains the privacy of all participants.

3.1.3 Decentralized and Resilient Data Backup

Protecting sensitive data through backups is crucial. Instead of entrusting data to a single commercial provider, users might prefer to create encrypted backups, break them into fragments, and distribute these fragments across a network of trusted peers or their own devices in different locations. This approach primarily aims to enhance data durability against loss.

3.1.4 Ad-hoc Sharing of Personal Files

Users frequently have personal digital files, for instance, group photographs from an event, family videos, or personal documents, that they wish to share quickly and privately with specific individuals or a small group of trusted contacts. In such scenarios, the primary need is a straightforward and secure mechanism to select specific files or directories from one's personal collection and transmit them directly only to the intended recipients, ensuring that others cannot access the shared content.

3.1.5 Blob Storage for Applications

Applications often store *blobs*, such as images, documents, etc., in various centralized storage backends, or even simply in the local filesystem of their host machine. Using a decentralized data sharing system as a storage backend would allow them to benefit from its built-in redundancy and high availability, thus drastically reducing the cost of creating backups.

3.2 Key Requirements

Based on the aforementioned illustrative use cases, a set of fundamental requirements can be delineated for any protocol aiming to facilitate secure data sharing. These requirements serve as essential criteria for the design and evaluation of such a system, ensuring it is robust, trustworthy, and fit for purpose:

3.2.1 Confidentiality

The protocol must ensure that shared data is accessible only to authorized participants. Data, whether in transit across the network or at rest on participating peers (including in termediate storage nodes, if any), must be protected from unauthorized disclosure

through strong cryptographic mechanisms. [42] This implies that individuals or entities not explicitly granted access by the data owner or an authorized party cannot decipher or view its contents.

3.2.2 Integrity

The protocol must guarantee that data remains unaltered during transmission and storage. Participants need assurance that the data they receive or retrieve is identical to the data originally sent or stored by the legitimate source.

3.2.3 Availability

The protocol must ensure that shared data is accessible to authorized users when they request it. In a peer-to-peer network, where nodes can be transient and network conditions may vary, maintaining high availability is a significant challenge. [43] The system should employ strategies to mitigate the impact of individual node unavailability, aiming to provide consistent access to data for legitimate participants despite the dynamic nature of the network. This implies that data should not become entirely inaccessible if a large subset of nodes becomes unreachable.

3.2.4 Durability

Beyond immediate availability, the protocol must ensure the long-term persistence of shared data, protecting it against loss due to node failures, departures from the network, or data corruption. Durability implies that data, once committed to the system by an authorized party, should remain intact and retrievable over extended periods, even in the face of adversarial conditions and the natural churn of participating peers. Mechanisms such as data redundancy, replication across multiple nodes, or error-corection coding (ECC) are typically essential for achieving robust data durability in a decentralized environment.

3.2.5 Scalability

The protocol must be designed to scale effectively with an exponentially increasing number of participants, an ever-growing aggregate of shared data, and rising network traffic. Scalability in a peer-to-peer context implies that the system's performance (e.g., data retrieval times, discovery latency, overall throughput) should not degrade disproportionately as the network expands. [43] The design should avoid centralized bottlenecks and distribute load in a manner that accommodates growth. For a data sharing protocol, this means efficiently dividing the body of data among all peers in a manner that prevents any one peer from being overloaded.

3.2.6 Efficiency

The protocol should operate efficiently to minimize resource consumption. This encompasses several aspects:

- **Computational Efficiency:** Cryptographic operations, data processing, and protocol logic should be designed to minimize CPU load, enabling participation from a wide range of devices, including those with limited processing power.

- **Bandwidth Efficiency:** The protocol should minimize the amount of data transmitted over the network. This includes optimizing the size of protocol messages, avoiding unnecessary data transfers, and potentially employing techniques for data compression or differential updates where applicable.
- **Storage Efficiency:** For peers contributing storage, the protocol should aim to minimize the storage overhead associated with metadata, redundancy schemes, or indexing structures, while still meeting durability requirements.

Overall efficiency is crucial for user adoption, particularly in resource-constrained environments or for large-scale data sharing, as excessive resource demands can deter participation and degrade system performance.

3.2.7 Developer Usability and Ease of Integration

For the protocol to achieve widespread adoption and be correctly implemented in diverse applications, it must be designed with developer usability in mind. This implies several considerations:

- **Clear Specification:** The protocol specifications should be comprehensive, unambiguous, and easy to understand.
- **Simple Application Programming Interfaces (APIs):** APIs should be intuitive, well-documented, and designed to minimize the likelihood of common security pitfalls or misuse by application developers [18].
- **Modularity and Extensibility:** A modular design can simplify integration with existing systems and allow for future extensions or adaptations without requiring a complete overhaul.
- **Availability of Reference Implementations or Libraries:** The existence of well-maintained reference implementations or robust software development kits (SDKs) can significantly lower the barrier to entry for developers and promote consistent, secure usage.

Ultimately, a protocol that is difficult to integrate or prone to misconfiguration by developers is less likely to be adopted and used securely, regardless of its theoretical strengths.

3.3 Conclusion

This chapter has explored illustrative use cases for secure peer-to-peer data sharing, providing a context for understanding the diverse needs and motivations that drive the demand for such systems. Based on this exploration and fundamental principles of secure system design, a core set of requirements has been delineated. These essential requirements include ensuring robust data **confidentiality** (3.2.1) to protect information from unauthorized access, and guaranteeing data **integrity** (3.2.2) to prevent illicit modifications.

Furthermore, the system must provide high **availability** (3.2.3), ensuring data is accessible to authorized users when needed, and **durability** (3.2.4), safeguarding data against loss over time despite node failures. The protocol must also be designed for **scalability** (3.2.5), allowing it to perform effectively as the network of participants and the volume of data grow. **Efficiency** (3.2.6) in terms of computational, bandwidth, and storage resources is crucial for practical adoption. Finally, **developer usability and ease**

of integration (3.2.7) are paramount to facilitate correct and widespread implementation in various applications.

These identified requirements collectively form a foundational framework. They will serve as guiding principles and critical benchmarks for the design and subsequent evaluation of the secure peer-to-peer data sharing protocol detailed in the following chapters of this thesis. Adherence to these requirements is vital for creating a system that is not only secure and functional but also practical and trustworthy for its users.

Chapter 4

Secure Data Serialization Format

This chapter specifies the **ScatterNet** secure data serialization format, and defines its precise structure and cryptographic transformations applied to data units, thus forming the foundation for secure data exchange within the protocol.

4.1 The DataChunk: Fundamental Unit of Data

The **DataChunk** is the foundational unit of data in `ScatterNet`. It consists of arbitrary byte sequence `data` and its cryptographic `hash`, which facilitates secure handling and addressing.

4.1.1 Conceptual Definition and Purpose

A `DataChunk` conceptually represents any discrete piece of data that users wish to securely store or exchange through the `ScatterNet` system. The purpose of the `DataChunk` abstraction is to provide a uniform mechanism for protecting, addressing, and verifying arbitrary data units regardless of their content or significance.

4.1.2 Logical Composition

For any raw input data D, the system calculates $H(D)$ using the hash function defined in Section 4.2, producing a 64-character[1] hash value. This hash serves multiple critical purposes:

- It functions as the symmetric encryption key when encryption is applied.
- It forms part of the `HKey` (Section 4.5) used for addressing.
- It enables content verification after decryption.

The hash computation establishes the cryptographic foundation for all subsequent operations on the data.

[1] `Hash` is composed of **64** `Base64` characters, see Section 4.2.

4.1.3 Serialized Form of a DataChunk

The serialized form of a `DataChunk` consists of the concatenation of its `hash` and `data`:

```
SerializedDataChunk = H(data) || data
```

where $H(data)$ is the 64-byte ASCII representation of the 64-character `Base64` string of the `data`'s `hash` (as defined in Section 4.2), and `||` denotes concatenation. The first 64 bytes of the serialized form are therefore a `Base64` string.

4.2 The Hash: a Universal Reference

The `Hash` identifier is a critical component in `ScatterNet`, providing a unique, verifiable, and robust fingerprint for data. It is a 64-character Base64 string derived from a combination of cryptographic hashes, the data's length, and Reed-Solomon parity, ensuring both content integrity and resilience against corruption of the identifier itself.

4.2.1 Choice of Algorithms: SHA-256 and BLAKE3

The core of the `Hash` construction employs two distinct cryptographic hash functions: SHA-256 and BLAKE3.

- **SHA-256:** A member of the SHA-2 family, SHA-256 is a widely adopted and well-analyzed algorithm, standardized by NIST. [34] Its established security and broad acceptance make it a conservative choice for ensuring data integrity. It produces a 256-bit (32-byte) hash value.
- **BLAKE3:** A modern cryptographic hash function designed for high performance, strong security, and parallelizability. It offers significant speed advantages on contemporary hardware while maintaining a high security margin. It also produces a 256-bit, 32-byte output.

The combination of these two distinct algorithms provides defense in depth: should a cryptographic weakness be discovered in one, the other is intended to maintain the overall security of the combined hash.

4.2.2 Composition and Structure

The `Hash` identifier in its binary form is a 48-byte sequence constructed as follows:

Core Hash: SHA-256 $\oplus$ BLAKE3

The initial 32-byte segment is generated by computing the SHA-256 hash of the input data, D, and the BLAKE3 hash of the same data. The 32-byte output from the BLAKE3 algorithm is then bitwise XORed with the 32-byte SHA-256 hash:

$$\text{CoreHash} = \text{SHA256}(D) \oplus \text{BLAKE3}(D) \tag{4.1}$$

This results in a 32-byte combined cryptographic digest.

Appending Length Information

Following the CoreHash, a 2-byte representation of the original data's approximate length is appended. This length information is encoded using the `PackedInt` format (see Section 4.2.4).

$$\text{BinaryHash}_{\text{partial}} = \text{CoreHash} \mid\mid \text{PackedInt}(\text{len}(D)) \tag{4.2}$$

At this stage, the binary hash is 34 bytes long.

Appending Reed-Solomon Parity

To enhance the robustness of the hash identifier against corruption, Reed-Solomon error correction is applied. A Reed-Solomon code RS(48, 34) is used, generating 14 bytes of parity data. These parity bytes are calculated over the 34-byte sequence from the previous steps and then appended to form the complete 48-byte binary hash:

$$\text{BinaryHash}_{\text{complete}} = \text{CoreHash} \mid\mid \text{PackedInt}(\text{len}(D)) \mid\mid \text{ReedSolomonParity} \tag{4.3}$$

This Reed-Solomon configuration can correct up to 7 corrupted bytes within the 48-byte structure. Further details on the Reed-Solomon codes are provided in Section 4.3.

Base64 Encoding to Hash Value

The complete 48-byte binary hash is transformed into its final form through Base64 encoding. This encoding process converts the 384 bits (48 bytes × 8 bits/byte) of binary data into a 64-character ASCII string, with each character representing 6 bits of the original data. The encoding uses the encoding variant described in Section 4.2.3.

$$\text{Hash} = \text{Base64Encode}(\text{BinaryHash}_{\text{complete}}) \tag{4.4}$$

This 64-character string constitutes the final Hash value that serves as the canonical identifier throughout the `ScatterNet` system.

4.2.3 Custom Base64 Encoding Scheme

The `ScatterNet Hash` utilizes a custom Base64 encoding scheme to transform its 48-byte binary form into a 64-character ASCII string. Base64 encoding, in general, is a method for representing binary data in an ASCII string format by translating it into a radix-64 representation, where each character represents 6 bits of the original data. The specific variant used in `ScatterNet` is designed with considerations for URL safety, character properties, and compatibility.

Rationale for Custom Alphabet Selection

The 64 characters constituting the `ScatterNet` Base64 alphabet are selected based upon a systematic application of criteria designed to ensure URL safety and define a consistent character set. The process is as follows:

1. **Initial Character Pool – URL Unreserved Characters:** The foundation for the alphabet is the set of „unreserved characters“ defined in RFC 3986 [5].

This set comprises uppercase letters (`A-Z`), lowercase letters (`a-z`), digits (`0-9`), hyphen (`-`), period (`.`), underscore (`_`), and tilde (`~`). These characters are generally not subject to percent-encoding in URIs.

2. **ASCII Value Constraint:** A filter is applied to this initial pool: only characters whose 8-bit ASCII value is greater than or equal to the ASCII value of the character `'0'` (decimal 48) are retained. This constraint results in the following 64-character set:
   - Digits: `0-9`
   - Uppercase letters: `A-Z`
   - Underscore: `_`
   - Lowercase letters: `a-z`
   - Tilde: `~`
3. **Alphabet Ordering for Mapping:** The specific sequence in which these 64 selected characters are arranged defines the mapping from 6-bit input values (0-63) to their character representations. The alphabet order is chosen to align with the alphanumeric sequence common to standard Base64 [25] and Base64url [25], with tilde and underscore appended to complete the set.

Alphabet Definition and Mapping

Based on the aforementioned rationale, the `ScatterNet` Base64 alphabet is defined by the following 64 characters, ordered to map to input values 0 through 63:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789~_
```

This specific ordering dictates the mapping: the first 6-bit value (0) maps to `A`, the second (1) to `B`, and so on, with the 62nd value mapping to `9`, the 63rd value mapping to `~`, and the 64th value mapping to `_`.

Padding Character Omission

Standard Base64 encoding may append one or two padding characters (`=`) at the end of the encoded string if the number of input bytes is not a multiple of three. The Implementation shall not print padding characters, and shall ignore them during decoding.

4.2.4 PackedInt Encoding Format

The `PackedInt` format is a compact, 16-bit (2-byte) scheme used within `ScatterNet` for representing non-negative integer values, primarily for encoding approximate lengths. It consists of an 8-bit exponent byte (E) followed by an 8-bit mantissa byte (M), forming the structure: ($E \mid\mid M$).

The value represented by a `PackedInt` is decoded as follows:

- If the exponent byte $E < 2$: The value is given by Value $= (E \ll 8) + M$. This directly covers integers from 0 (represented as `0x0000`) up to 511 (represented as `0x01FF`).

- If the exponent byte $E \geq 2$: The value is calculated as Value $= 2^{(E+7)} + (M \times 2)$. For example, the value 512 is represented as `0x0200`. Subsequent values are represented with a step of 2 for each increment of the mantissa M.

This encoding allows `PackedInt` to represent a wide range of magnitudes. For values less than 512, it provides exact representation with a precision of 1. For values greater than or equal to 512 (i.e., when $E \geq 2$), the precision becomes 2, meaning only even integers can be represented, aligning with its use for indicating approximate lengths where exact precision for very large numbers is not critical.

4.3 Reed-Solomon Codes for Error Correction

Reed-Solomon (RS) codes are a class of non-binary cyclic error-correcting codes renowned for their ability to correct multiple symbol errors. [38] An RS code is typically defined by parameters (N,K), where K is the number of data symbols and N is the total number of symbols in an encoded block (codeword), with N-K parity symbols appended. Such a code can correct up to ($t = \lfloor (N-K)/2 \rfloor$) erroneous symbols [30]. This section details how `ScatterNet` utilizes Reed-Solomon codes in two distinct ways: a direct application for protecting the `Hash` identifier, and a more complex scheme, termed Long Error Correction Code (Long ECC), for ensuring the resilience of encrypted `DataChunk`s.

`ScatterNet` uses RS codes in GF(2^8) with a symbol size of 1 byte.

4.3.1 Direct RS Application to `Hash`es

To protect the integrity of `Hash`es, `ScatterNet` employs a direct application of the Reed-Solomon code RS(48, 34). [4.2.2]

- The message for this code consists of the first 34 bytes of the `hash` in binary form.
- From this 34-byte message, 14 parity bytes are generated.
- These 14 parity bytes are appended to the 34-byte message, forming the complete 48-byte `BinaryHash_complete` with the structure: `Message (34B) || Parity (14B)`.
- Within the Reed-Solomon codeword itself, however, the 14 parity symbols correspond to symbols 0–13, while the 34 message symbols follow.

This configuration, operating over GF(2^8) where each byte is a symbol, allows for the correction of up to 7 byte errors within the 48-byte binary hash.

4.3.2 Advanced Application: Long ECC for `DataChunk` Payload

For the protection of the encrypted `DataChunk`s, `ScatterNet` implements a custom Long Error Correction Code (Long ECC) scheme. This scheme is built upon Reed-Solomon coding principles and incorporates a self-correcting header to manage the error correction process for the main data.

The `LongEccHeader` and its Self-Correction

The Long ECC scheme is managed by a 16-byte `LongEccHeader`. To ensure the reliability of its critical parameters, the descriptive fields within this header are themselves protected by a Reed-Solomon code. The 16-byte header is structured with its descriptive fields first, followed by their parity symbols:

- **Header Fields (12 bytes):** The initial 12 bytes contain the `LongEccHeader` descriptive fields:
  - `full_length: u32` (Total length of the codeword)
  - `message_length: u32` (Length of the `encrypted_payload`)
  - `parity: u8` (The maximum number of correctable mistakes in a segment)
  - `segment_length: u8`
  - `segment_distance: u8`
  - `last_segment_length: u8`
- **Header Parity (4 bytes):** The subsequent 4 bytes are Reed-Solomon parity symbols calculated over the preceding 12 bytes of header fields.

This header structure utilizes an RS(16, 12) code, where the 12 bytes of header fields constitute the message, and the 4 appended parity symbols protect these fields. This allows for the correction of up to $t = (16 - 12)/2 = 2$ byte errors. The overall structure is `HeaderFields (12B) || HeaderParity (4B)`.

Interleaved Parity Scheme for Payload Data

The `encrypted_payload` (which follows the 16-byte `LongEccHeader`) is protected by an interleaved Reed-Solomon parity scheme, governed by the parameters within the `LongEccHeader`. Parity symbols are generated for segments of the data to be protected (i.e., the `encrypted_payload`, and potentially the `LongEccHeader` fields themselves if they are also included in this stage of protection).

The conceptual process for generating the main payload parity is as follows:

1. The data to be protected is processed in segments. The first segment starts at the first byte of the header.
2. For each segment of `segment_length` (or `last_segment_length` for the final segment), Reed-Solomon parity symbols are calculated. The number of parity symbols per segment is $2 * parity$, where *parity* is the value of the `parity` field in the header.
3. These generated parity symbols are appended to the codeword.
4. The starting point for the next segment is determined by advancing the current position by `segment_distance`.

This iterative process continues until the entire codeword has been processed.

The final structure of the Long ECC codeword is:

$$\texttt{Codeword} = \texttt{LongEccHeader (16B)} \,||\, \texttt{encrypted\_payload} \,||\, \texttt{parity bytes} \tag{4.5}$$

The `full_length` field in the header specifies the total length of this `EncryptedData`.

4.3.3 Configuration Summary

The Reed-Solomon error correction configurations employed within the `ScatterNet` serialization format are summarized below. All RS codes operate over GF(2^8), treating each byte as a symbol.

- **`Hash` Identifier Protection (Direct RS Application):**
  - **Code:** RS(48, 34).
  - **Message Length:** 34 bytes (comprising the 32-byte CoreHash and 2-byte PackedInt length).
  - **Parity Length:** 14 bytes.
  - **Codeword Structure (Binary `Hash` Form):** Systematic, `Message (34B) || Parity (14B)`.
  - **RS Codeword Symbol Order:** Parity symbols occupy positions 0–13, followed by message symbols.
  - **Error Correction Capability:** Up to 7 byte errors within the 48-byte binary hash.

- **Encrypted `DataChunk` Protection (Long ECC Scheme):**
  - **`LongEccHeader` Self-Correction:**
    - **Code:** RS(16, 12).
    - **Message Fields Length:** 12 bytes (the descriptive fields of the `LongEccHeader`).
    - **Parity Length:** 4 bytes.
    - **Header Structure:** `HeaderFields (12B) || HeaderParity (4B)`.
    - **Error Correction Capability:** Up to 2 byte errors within the 12-byte header fields.
  - **Overall Data Protection (Iterative Interleaved RS for Long ECC):**
    - **Process Overview:** The `AppendedInterleavedParity` is generated via an iterative, interleaved Reed-Solomon process, governed by parameters in the `LongEccHeader`.
    - **Segment Data Source:** For each iteration, a data segment is processed. This segment is drawn from a conceptual „current codeword“ which consists of the initial `LongEccHeader`, the `ChaChaEncrypted` payload, and all Reed-Solomon parity symbols generated and appended from *prior* segments in the iteration.
    - **Parity Generation per Segment:** RS parity is calculated for the current segment. These newly generated parity symbols are then appended, extending the „current codeword“ for subsequent iterations.
    - **Non-Recursive Termination:** The parity symbols generated for the final data segment processed in the iteration are not themselves included as input into any subsequent parity calculation step within this scheme.
    - **Final Parity Block:** The `AppendedInterleavedParity` is the total collection of all parity symbols generated throughout this iterative process.

* **Final Long ECC Codeword Structure:** `LongEccHeader (16B) || ChaChaEncrypted || AppendedInterleavedParity`. This entire structure constitutes the `EncryptedData` payload.
* **Error Correction Capability:** Determined by the `LongEccHeader.parity` field (defining parity symbols per segment) and the RS parameters applied to each segment.

These configurations provide layered error correction for different components of the `ScatterNet` data structures, with the Long ECC employing an iterative method to protect the combined header, payload, and incrementally generated parity.

4.4 `DataChunk` Encryption and Integrity

To ensure the confidentiality and integrity of shared data, `ScatterNet` employs an authenticated encryption scheme. This section details the process of transforming a `SerializedDataChunk` into a protected, encrypted form, suitable for storage and transmission within the peer-to-peer network. The overall encryption pipeline is illustrated in Figure 4.1.

4.4.1 Overview of the Encryption Process

The encryption process takes a `SerializedDataChunk` (as defined in Section 4.1.3), compresses it, and then encrypts the compressed data using the ChaCha20-Poly1305 AEAD scheme. The encryption key is deterministically derived from the uncompressed `SerializedDataChunk` itself. The resulting encrypted data, along with this key, forms the basis for the `HKey` (detailed in Section 4.5), which is used for addressing and retrieving the encrypted content.

4.4.2 Data Pre-processing for Encryption

Before encryption, the `SerializedDataChunk` undergoes a compression step.

Initial Data Form: The `SerializedDataChunk`

The input to the encryption pipeline is the `SerializedDataChunk`, which, as defined in Section 4.1.3, consists of the concatenation of the `Hash` of the original raw data and the raw data itself:

$$\text{SerializedDataChunk} = H(\text{raw\_data}) \,||\, \text{raw\_data} \tag{4.6}$$

Compression

The `SerializedDataChunk` is compressed using the Deflate algorithm [14].

$$\text{DeflatedChunk} = \text{Deflate}(\text{SerializedDataChunk}) \tag{4.7}$$

Compression is applied prior to encryption to reduce the overall size of the data to be encrypted and subsequently stored or transmitted. This improves storage efficiency and reduces bandwidth consumption.

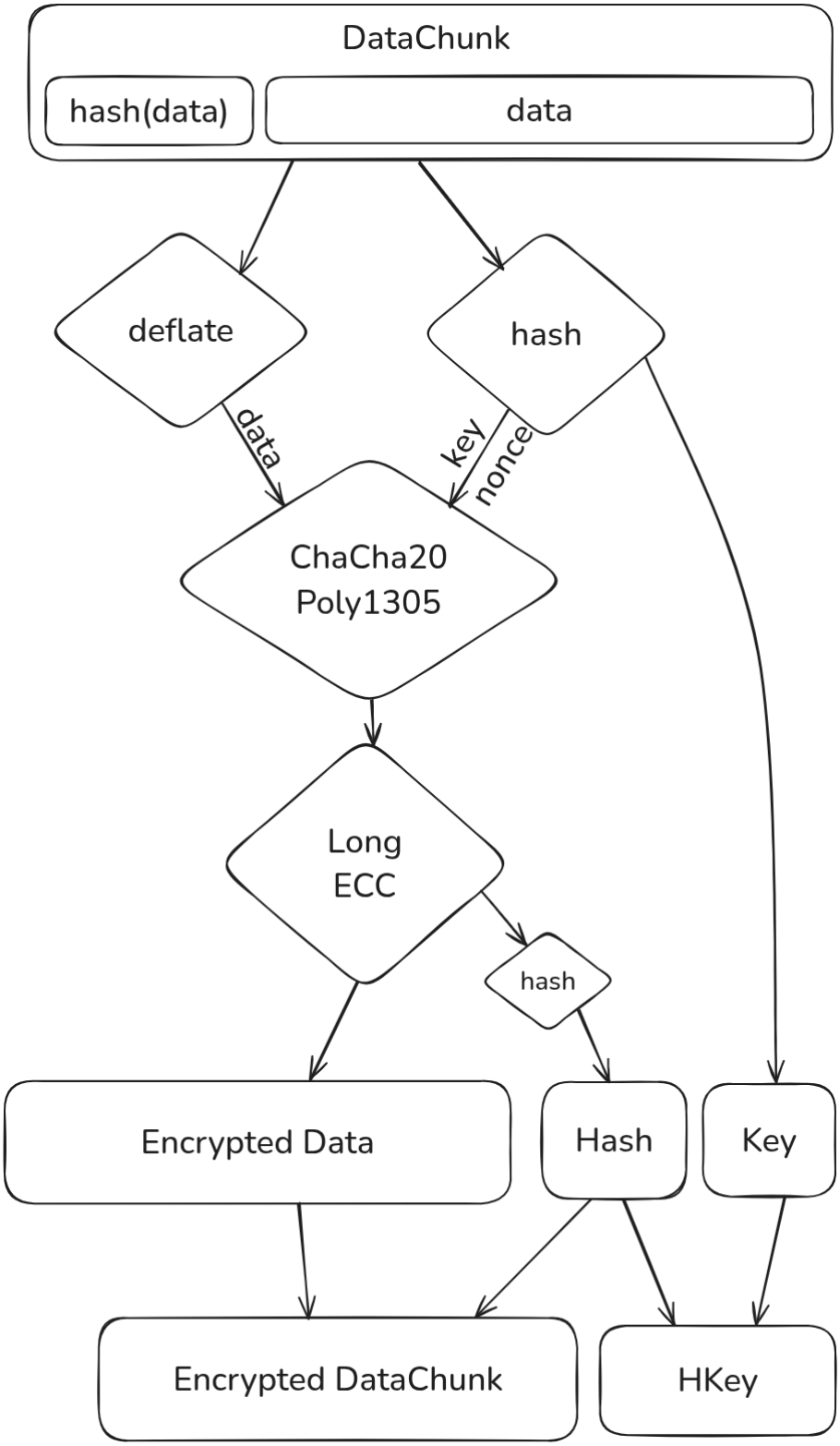

Figure 4.1: ScatterNet DataChunk Encryption Scheme.

4.4.3 Encryption Key Derivation

The symmetric encryption key for the AEAD scheme is derived by computing the `Hash` (as defined in Section 4.2) of the original, uncompressed `SerializedDataChunk`:

$$\text{EncryptionKey} = H(\text{SerializedDataChunk}) \tag{4.8}$$

This ensures that the key is deterministically tied to the specific content of the `SerializedDataChunk` (which includes both the raw data and its own hash). This`EncryptionKey` is one of the two components of the `HKey`.

4.4.4 AEAD Encryption

`ScatterNet` utilizes the ChaCha20-Poly1305 algorithm as its Authenticated Encryption with Associated Data (AEAD) scheme.

Selected Scheme: ChaCha20-Poly1305

ChaCha20-Poly1305 was selected for its strong security profile and excellent performance in software implementations, particularly on a wide range of CPU architectures, as discussed in Section 2.3. It provides both confidentiality for the data through ChaCha20 encryption and integrity and authenticity through the Poly1305 message authentication code.

Encryption Operation

The encryption operation proceeds as follows:

- **Plaintext:** The `DeflatedChunk` is used as the plaintext input.
- **Key:** The `EncryptionKey` (which is H(SerializedDataChunk), as derived in Section 4.4.3) is used as the 256-bit key for ChaCha20-Poly1305.
- **Nonce:** A 96-bit (12-byte) nonce is required. This nonce is derived from the `EncryptionKey`. Specifically, the `EncryptionKey` is a `Hash` whose 48-byte binary form consists of a 32-byte CoreHash, a 2-byte PackedInt length, and 14 bytes of Reed-Solomon parity (as laid out in detail in Section 4.2.2). The last 12 bytes of this 14-byte Reed-Solomon parity portion of the `EncryptionKey`'s binary form are used as the nonce for the ChaCha20-Poly1305 operation. This deterministic derivation ensures the nonce is tied to the key, while the properties of the hash construction aim to provide sufficient variability.
- **Associated Data (AAD):** No Associated Data is utilized; the AAD input to ChaCha20-Poly1305 is empty. The integrity protection is focused on the ciphertext itself.

The output of the AEAD encryption is a buffer containing the ciphertext and the Poly1305 authentication tag:

$$\text{ChaChaEncrypted} = \text{ChaCha20Poly1305}(klic, nonce, \varepsilon, DeflatedChunk) \tag{4.9}$$

4.4.5 Application of Long Error Correction Code

The `ChaChaEncrypted` buffer, produced by the ChaCha20-Poly1305 AEAD scheme, is then further processed by the Long Error Correction Code (Long ECC) scheme detailed in Section 4.3.2.

$$\text{EncryptedData} = \text{LongECCEncode}(\text{ChaChaEncrypted}) \tag{4.10}$$

This step adds a layer of resilience against physical data corruption to the already encrypted and authenticated data. The `EncryptedData` is the data's final form: in this form it shall be both stored and transmitted.

4.5 The `HKey`: Universal Data Identifier and Access Key

The `HKey` (HashKey) is a universal identifier within `ScatterNet` used to address, retrieve, and facilitate the verification or decryption of various forms of data. It provides a standardized string representation that encapsulates not only the identity of a piece of data but often also the means to access its content, whether it be raw, encrypted, or a complex aggregation of other data units.

4.5.1 Definition and General Structure

An `HKey` is a string composed of a single-character prefix indicating its type, followed by type-specific data. This prefix allows `ScatterNet` peers to correctly parse the `HKey` and understand the nature of the data it references. The general structure is:

$$\texttt{HKey} = \text{PrefixCharacter} \,||\, \text{TypeSpecificData} \tag{4.11}$$

The following subsections detail the defined `HKey` variants. The term `Hash` used in these definitions refers to the 64-character Base64 string produced by the `ScatterNet` hash function (Section 4.2), and `Key` typically refers to an `EncryptionKey` (also a `Hash`) as derived in Section 4.4.3.

4.5.2 `HKey` Variants

`ScatterNet` defines several `HKey` variants to accommodate different data types and storage strategies:

Base64 Variant ('B' prefix)

This variant is designed for embedding very small pieces of raw, unencrypted data directly within the `HKey` string itself.

- **Structure:** `'B' || Base64(rawData)`
- **Components:**
  - `'B'`: The single-character prefix.
  - `Base64(rawData)`: The Base64 encoding (Section 4.2.3) of the `rawData`.
- **Purpose:** To efficiently represent and transmit small data segments without requiring a separate data retrieval step.

- **Constraint:** This variant may be used for `rawData` up to 46 bytes in length. This ensures that the resulting Base64 encoded string is at most 62 characters, leading to a total `HKey` length of 63 characters, to prevent representation collisions with `Hash`es.

Direct Variant ('D' prefix)

This variant directly references an unencrypted `DataChunk` (or more precisely, its raw data component) using its `Hash`.

- **Structure:** `'D' || HashOfRawData`
- **Components:**
  - `'D'`: The single-character prefix.
  - `HashOfRawData`: The `Hash` calculated directly over the unencrypted raw data (i.e., H(raw_data) as per Section 4.1.2).
- **Purpose:** To address and retrieve data that is stored in its raw, unencrypted form within the `ScatterNet` system. The referenced data is expected to be the `raw_data` part of a `SerializedDataChunk` whose prefix matches `HashOfRawData`.
- This variant's prefix is **Optional**. This means every `Hash` is a valid `HKey`.

Encrypted Variant ('E' prefix)

This is the default variant for referencing an encrypted `DataChunk` payload, as processed by the encryption pipeline detailed in Section 4.4.

- **Structure:** `'E' || HashOfEncrypted || EncryptionKey`
- **Components:**
  - `'E'`: The single-character prefix.
  - `HashOfEncrypted`: The `Hash` of the final `EncryptedData` (see Section 4.4.5). This `EncryptedData` is the output of the Long ECC scheme applied to the AEAD-encrypted data.
  - `EncryptionKey`: The `Hash` used as the key for the AEAD encryption (i.e., H(SerializedDataChunk) as per Section 4.4.3).
- **Purpose:** To address, retrieve, and provide the decryption key for an encrypted `DataChunk` payload. This is the standard `HKey` form resulting from the full data protection pipeline.

List Reference Variant ('L' prefix)

This variant provides an indirect reference to an encrypted `DataChunk` that itself contains the textual representation of a `List` Variant `HKey` (Section 4.5.2) or a `Long` Variant `HKey` (Section 4.5.2).

- **Structure:** `'L' || Hash || Key` (see Section 4.5.2)
- **Components:**

- ‘L’: The single-character prefix.
- Hash: The Hash of the encrypted representation of a string that defines a List or Long HKey.
- Key: The Hash used as the key to decrypt the aforementioned string.

- **Purpose:** To efficiently reference potentially large or complex List or Long HKey definitions that are stored as encrypted DataChunks within ScatterNet. This avoids embedding very long strings directly into other HKeys or metadata. Upon retrieval and decryption, the content is parsed as a List or Long HKey.

List Variant (‘[’ prefix)

This variant defines a logical data object composed of the concatenation of data referenced by a sequence of other HKeys.

- **Structure:** ‘[’ || $\texttt{HKey}_1$ || ‘,’ || $\texttt{HKey}_2$ || ... || ‘,’ || $\texttt{HKey}_n$ || ‘]’
- **Components:**
  - ‘[’ and ‘]’: Delimiting characters.
  - $\texttt{HKey}_i$: A comma-separated sequence of one or more valid HKey strings (which can be of any variant, including other List or Long variants, allowing for nesting).
- **Purpose:** To represent a data object formed by sequentially concatenating the complete data content referenced by each constituent HKey. For example, if $\texttt{HKey}_1$ references data D_1 and $\texttt{HKey}_2$ references data D_2, this List Variant represents $D_1||D_2$.

Long Variant (‘{’ prefix)

This variant defines a logical data object assembled from potentially partial segments of data referenced by other HKeys, allowing for the construction of larger logical files from smaller, possibly non-contiguous, pieces.

- **Structure:**
  ‘{’ || $\texttt{segment}_1$ || ‘,’ || $\texttt{segment}_2$ || ... || ‘,’ || $\texttt{segment}_n$ || ‘}’
- **Components:**
  - ‘{’ and ‘}’: Delimiting characters.
  - $\texttt{segment}_i$: A comma-separated sequence of segment definitions. Each segment can be of two forms:
    1. An HKey string: This implies the entire data content referenced by this HKey forms a contiguous segment in the logical object.
    2. A range-specified HKey: startByte-endByte:HKeyString. This specifies that the segment consists of the byte range from startByte to endByte (inclusive) of the data referenced by HKeyString.
- **Purpose:** To define large or complex data structures by aggregating entire data chunks or specific byte ranges from various constituent data chunks. For example, {0-4095:HKeyA,HKeyB,8192-12287:HKeyC} would construct a logical file where the first 4096 bytes are from HKeyA, followed by the entire content of HKeyB, followed by bytes 8192 through 12287 from HKeyC.

Chapter 5

Communication Protocol

This chapter specifies the peer-to-peer communication protocol for the `ScatterNet` system. Building upon the secure data serialization format defined in Chapter 4, this protocol outlines the fundamental interactions required for participants to securely store and retrieve data chunks. The primary goal is to define a clear, minimal, and robust interface that enables decentralized data sharing while abstracting away the complexities of underlying network transport mechanisms. This chapter will detail participant identification, the core operational interface, and the expected semantics of message exchanges.

5.1 Core Protocol Principles

The `ScatterNet` protocol is designed with several core principles to ensure security, interoperability, and clarity of scope.

5.1.1 Participant Identification

Each participant (peer) in the `ScatterNet` network is uniquely identified by their Ed25519 public key [24]. Ed25519, an Edwards-curve Digital Signature Algorithm (EdDSA) scheme using Curve25519, is chosen for its strong security properties, high performance, resistance to many side-channel attacks, and compact key and signature sizes. The public key serves as the persistent identifier for a peer, while the corresponding private key enables the peer to authenticate its actions, such as signing messages or proving ownership of stored data, should higher-level protocol extensions require such features. For the basic interface defined herein, the public key primarily serves as an addressable identity.

5.1.2 Protocol Layering and Scope

The `ScatterNet` protocol defined in this chapter specifies a logical communication interface and the semantics of operations between peers. It intentionally does not dictate the specifics of:

- **Network Transport:** Peers are free to implement the defined interface over any suitable network transport layer (e.g., TCP/IP, UDP, QUIC, or other peer-to-peer overlay networks).

- **Packet Structure:** The on-the-wire format of packets used to convey protocol messages is considered an implementation detail. Implementations must agree on a common packet structure if they are to interoperate, but this is not part of the core `ScatterNet` protocol specification itself.
- **Peer Discovery and Routing:** Mechanisms for discovering other peers or routing requests through the network are considered separate concerns, potentially handled by an underlying P2P library or a dedicated overlay network.

This layered approach allows for flexibility and adaptability, enabling `ScatterNet` to be deployed in diverse network environments. The focus of this specification is the application-level interface for data interaction.

5.2 Interface Definition

Each participant in the `ScatterNet` network must implement a minimal, well-defined interface to enable the fundamental operations of data storage and retrieval. This interface consists of two primary operations: `FETCH` and `PUT`.

5.2.1 `FETCH` Operation

Purpose

The `FETCH` operation is used by a requesting peer (client) to retrieve a protected data payload from a responding peer (storer) that is presumed to hold it.

Parameters

- **Input:**
  - `hash`: A `Hash` (as defined in Section 4.2). This `hash` corresponds to the `HashOfEncrypted` component of an `HKey` (see Section 4.5.2), which is $H(\text{EncryptedData})$.

Return Values

- **Output (on success):**
  - `EncryptedData`: The requested data payload, which is the `EncryptedData` (as defined in Section 4.4.5). This payload includes the `LongEccHeader`, the AEAD-encrypted data (`ChaChaEncrypted`), and the `AppendedInterleavedParity`.
- **Output (on failure):**
  - `NotFound`: Indicates that the responding peer does not have data corresponding to the provided `hash`.
  - `Error`: Indicates a general error occurred during the processing of the request on the responding peer's side (e.g., internal storage error, temporary unavailability).

Semantics

Upon receiving a `FETCH` request, the responding peer attempts to locate the `EncryptedData` payload associated with the provided `hash`. If found, the peer transmits this payload back to the client. The client is then responsible for performing the necessary steps to verify and decrypt the payload using the full `HKey` (which it must possess independently), including Long ECC decoding and AEAD decryption. If the data is not found, or an error occurs, an appropriate status is returned.

5.2.2 `PUT` Operation

Purpose

The `PUT` operation is used by a client peer to request that a responding peer (storer) store a data payload. The primary mode of operation involves the client providing an already processed and protected data payload along with its corresponding `HKey`. Implementations may optionally also support a mode where raw data is provided, and the storer performs the full data protection pipeline.

Parameters (Primary Mode)

- **Input:**
  - `EncryptedData`: The fully processed and protected data payload, which is the `EncryptedData` (as defined in Section 4.4.5). This payload includes the `LongEccHeader`, the AEAD-encrypted data (`ChaChaEncrypted`), and the `AppendedInterleavedParity`.

Return Values

- **Output (on success):**
  - `Success` returns an `HKey` referring to the submitted data
- **Output (on failure):**
  - `LimitExceeded`: Indicates that the responding peer refused to store the data. See Section 5.3.
  - `Error`: Indicates a general error occurred during the storage of the data on the responding peer's side (e.g., storage failure).

Optional Raw Data Handling

Implementations of the `PUT` operation *may* also support an alternative input mode where the client provides:

- **Input (Optional Mode):**
  - `rawDataChunk`: An arbitrary byte sequence representing the raw data to be stored.

In this optional mode, the responding (storing) peer would perform the complete `ScatterNet` data protection pipeline as detailed in Chapter 4. If this optional mode is supported and successful, the return value is the `HKey`. Failure conditions (`LimitExceeded`, `Error`) remain applicable. The availability of this raw data handling mode is implementation-dependent.

5.3 Fair Use Policy and Reputation System

To promote equitable resource utilization and discourage abusive behavior within the `ScatterNet` network, participants (peers) shall implement a Fair Use Policy (FUP). This policy is enforced through a locally managed, subjective reputation system. While specific penalty magnitudes for certain actions and enforcement thresholds are implementation-dependent, the core mechanics of score adjustment are defined herein.

5.3.1 Reputation Score Mechanics

Each peer shall maintain its own independent reputation score for every other peer with which it has interacted. This score is a local assessment, reflecting the history of interactions from the perspective of the peer maintaining the score.

- **Initial Score:** When a peer (Peer X) interacts for the first time with another peer (Peer Y), Peer X shall assign Peer Y an initial reputation score of 0 in its local records.
- **Score Adjustments for `FETCH` Operations:**
  - When Peer A (client) successfully fetches a chunk from Peer B (storer), Peer A shall increment its locally stored reputation score for Peer B by +1.
- **Score Adjustments for `PUT` Operations:**
  - When Peer A (client) successfully requests Peer B (storer) to store a chunk, and Peer B accepts and stores the data (returning an `HKey`):
    - Peer A (client) shall increment its locally stored reputation score for Peer B (storer) by +3.
    - Peer B (storer) shall decrement its locally stored reputation score for Peer A (client) by -1.
- **Score Adjustments for Invalid Packets:**
  - If Peer X receives an invalid packet from Peer Y (e.g., malformed, violating protocol semantics beyond errors handled by defined operation return values), Peer X shall decrement its locally stored reputation score for Peer Y by an implementation-defined amount. This amount must be configurable and should be chosen to effectively discourage such behavior.

5.3.2 Enforcement of Storage Limits via Reputation

The primary mechanism for enforcing storage limits under the FUP is tied to reputation scores.

- Each peer implementation shall define a configurable negative reputation score threshold. A recommended default value is -2000, but peer operators must be able to adjust this.
- If a peer (storer) receives a `PUT` request (either primary mode or optional raw data mode) from another peer (client) whose reputation score, as maintained by the storer, is at or below this defined threshold, the storer shall refuse to store the data and shall return a `LimitExceeded` error.

This mechanism allows peers to protect their resources from disproportionate consumption by other peers.

5.3.3 Subjectivity and Scope of Reputation

The `ScatterNet` reputation system is designed with the following characteristics:

- **Local and Subjective:** All reputation scores are maintained locally by each peer and reflect that peer's direct interaction history and assessment of others. There is no central reputation authority or global score.
- **Implementation-Defined Parameters:** While the events triggering score changes are defined, the specific magnitude of decrement for „invalid packets“ and the exact „negative reputation score threshold“ for `PUT` refusal are implementation-dependent.

This design provides individual peers with a mechanism to manage resource usage and mitigate abuse based on direct interactions, without introducing the complexities of a global system.

Chapter 6

Implementation

This chapter details the reference implementation of the `ScatterNet` protocol and its constituent components. The implementation is realized as a collection of modular Rust crates, each responsible for a specific aspect of the system, from low-level data encoding and cryptographic primitives to the high-level peer logic. Rust was chosen as the implementation language for its strong emphasis on memory safety, performance, and concurrency, which are critical attributes for robust and efficient peer-to-peer systems. The following sections describe the key modules developed for `ScatterNet`, providing links to their documentation and source code repositories.

6.1 Core Library Crates

The foundational functionalities of `ScatterNet`, including data serialization, cryptography, and error correction, are encapsulated in a set of distinct Rust crates. This modular design promotes code reusability, maintainability, and testability.

6.1.1 `ps-base64`: Base64 Encoding

The `ps-base64` crate provides the custom Base64 encoding and decoding functionality utilized by `ScatterNet`, particularly for the generation of `Hash` identifiers. It implements the specific alphabet and padding rules detailed in Section 4.2.3. See the API documentation[1] and source code[2] for more.

6.1.2 `ps-pint16`: PackedInt Implementation

The `ps-pint16` crate implements the `PackedInt` 16-bit encoding scheme, as described in Section 4.2.4. This format is used for compactly representing approximate lengths within the `Hash` structure. See the API documentation[3] and source code[4] for more.

6.1.3 `ps-hash`: Cryptographic Hash Implementation

The `ps-hash` crate is responsible for generating the `ScatterNet Hash` identifiers. It implements the combined hashing scheme involving SHA-256 and BLAKE3, as in 6.1, length

[1] https://docs.rs/ps-base64
[2] https://github.com/prokopschield/ps-base64
[3] https://docs.rs/ps-pint16
[4] https://github.com/prokopschield/ps-pint16

encoding using `ps-pint16`, Reed-Solomon parity generation (leveraging `ps-ecc`), and the final Base64 encoding (using `ps-base64`), as specified in Section 4.2.

The hash algorithms for BLAKE3 and SHA-256 are imported from the well-known crates (libraries) `blake3` and `sha2`.

```
// The hash function (simplified) looks like this:
pub fn hash(data: &[u8]) -> Result<Hash, Error> {
   let mut buffer = Buffer::with_capacity(HASH_SIZE)?;

   buffer.extend_from_slice(sha256(data))?;
   buffer ^= &blake3(data).as_bytes()[..];
   buffer.extend_from_slice(PackedInt::from_usize(data.len()).to_16_bits())?;
   buffer.extend_from_slice(RS.generate_parity(&buffer)?)?;

   let hash = Self {
      inner: sized_encode::<HASH_SIZE>(&buffer),
   };

   Ok(hash)
}
```

Listing 6.1: Hash function implementation

See the API documentation[5] and source code[6] for more.

6.1.4 `ps-deflate`: Compression Utilities

The `ps-deflate` crate serves as a convenience wrapper around the `libdeflater` library, providing Deflate compression and decompression services. This is utilized in the data pre-processing stage before encryption, as outlined in Section 4.4.2. See the API documentation[7] and source code[8] for more.

6.1.5 `ps-ecc`: Error Correction Codes

The `ps-ecc` crate implements the Reed-Solomon error correction codes used in `ScatterNet`. This includes the RS(48, 34) code for `Hash` parity and the more complex Long Error Correction Code (Long ECC) scheme, including its self-correcting header and interleaved parity for `DataChunk` payloads, as detailed in Section 4.3. See the API documentation[9] and source code[10] for more.

6.1.6 `ps-datachunk`: DataChunk Representation

The `ps-datachunk` crate defines the `DataChunk` structure, which is the fundamental unit of data in `ScatterNet` (Section 4.1). It handles the logical composition of a `DataChunk` from raw data and its hash, as well as its serialized form.

[5] https://docs.rs/ps-hash
[6] https://github.com/prokopschield/ps-hash
[7] https://docs.rs/ps-deflate
[8] https://github.com/prokopschield/ps-deflate
[9] https://docs.rs/ps-ecc
[10] https://github.com/prokopschield/ps-ecc

`DataChunk` is defined as a trait[11], which directly offers methoods for serialization (`serialize`) and encryption (`encrypt`, `decrypt`). Structs built around specific underlying container types (e.g. `OwnedDataChunk`, `BorrowedDataChunk`, `SerializedDataChunk`) are also provided. See the API documentation[12] and source code[13] for more.

6.1.7 `ps-cypher`: Encryption Scheme Implementation

The `ps-cypher` crate implements the `DataChunk` encryption and integrity protection mechanisms detailed in Section 4.4. This includes key derivation from the `SerializedDataChunk`, AEAD encryption using ChaCha20-Poly1305 (using the `chacha20poly1305` crate), nonce management, and the application of the Long ECC scheme (via `ps-ecc`) to the encrypted output. See the API documentation[14] and source code[15] for more.

6.1.8 `ps-hkey`: `HKey` Processing and Data Access Abstraction

The `ps-hkey` crate defines the `HKey` structure (Section 4.5) and provides generic interfaces for its parsing, validation, and resolution. A key feature of this crate is its abstraction over the underlying data storage and retrieval mechanisms. Its methods accept closures that define how to fetch data given a `Hash` (e.g., `Fn(&Hash) -> Result<DataChunk, Error>`) or how to store data and obtain its `Hash` (e.g., `Fn(&[u8]) -> Result<Hash, Error>`). This design renders the `HKey` operations agnostic to the specific storage backend, allowing seamless interaction with diverse data sources such as local filesystem storage (e.g., via `ps-datalake`, Section 6.1.9) or remote network peers (e.g., via the `FETCH` and `PUT` operations defined in Section 5.2). Furthermore, `ps-hkey` implements the logic for resolving complex `HKey` variants, such as the List and Long variants, including capabilities for efficiently retrieving partial slices of the underlying data referenced by these composite structures. This crate is therefore central to interpreting `HKeys` and accessing the data they represent in a uniform manner. See the API documentation[16] and source code[17] for more.

6.1.9 `ps-datalake`: Filesystem Storage for DataChunks

The `ps-datalake` crate provides an interface for storing encrypted `DataChunks` into the local file systems, or directly onto memory media. It uses memory-mapped hash tables for organizing the data stored.

To calculate the ideal hash table index size, the crate uses a prime sieve that was adapted from Dave Plummer's Drag Race[18].

See the API documentation[19] and source code[20] for more.

[11] Traits are generic interfaces in the Rust programming language.
[12] https://docs.rs/ps-datachunk
[13] https://github.com/prokopschield/ps-datachunk
[14] https://docs.rs/ps-cypher
[15] https://github.com/prokopschield/ps-cypher
[16] https://docs.rs/ps-hkey
[17] https://github.com/prokopschield/ps-hkey
[18] https://github.com/PlummersSoftwareLLC/Primes/blob/drag-race/PrimeRust/solution_2/src/prime_object.rs
[19] https://docs.rs/ps-datalake
[20] https://github.com/prokopschield/ps-datalake

6.2 `scatter-net`: Peer Implementation and Application

The `scatter-net` crate serves as the reference implementation of a `ScatterNet` peer, providing both a library for programmatic integration and a standalone executable application. This crate orchestrates the functionalities of all the core library modules (detailed in Section 6.1) to realize the complete `ScatterNet` protocol specified in Chapter 5.

A key architectural choice in `scatter-net` is the utilization of the iroh peer-to-peer networking library for establishing and managing communication channels between peers. Iroh was selected for its design emphasis on simplicity and ease of use, coupled with its robust reliability and innovative approach to decentralized networking. It provides QUIC-based connections [23], offering secure, multiplexed, and low-latency transport suitable for `ScatterNet`'s interactive data exchange operations. The adoption of iroh also influences the asynchronous runtime environment; `scatter-net` employs the Tokio runtime, as it is the runtime utilized by iroh, ensuring seamless integration and efficient asynchronous operation.

Internally, `scatter-net` manages peer identity using Ed25519 keys and implements the server-side logic for the `FETCH` and `PUT` operations. This involves interaction with local storage abstractions (provided by `ps-datalake`) and, when acting as a storer in the optional raw data `PUT` mode, coordinating the full data protection pipeline. For the transmission of protocol messages over iroh's QUIC connections, `scatter-net` employs bincode for efficient binary serialization and deserialization of packet data, further optimized by applying compression using the `ps-deflate` crate. The crate also implements the client-side logic for initiating requests to other peers and manages the local reputation system as outlined in Section 5.3.

It is important to note that, in its current iteration, peer discovery mechanisms are not implemented within `scatter-net`; peers must be configured with the identifiers (public keys) of other known participants. The `scatter-net` implementation is envisioned as an evolving project, with plans for iterative enhancements and feature additions in the future to further expand its capabilities and robustness within the peer-to-peer ecosystem.

See the documentation[21] and the full source code[22] for more.

[21] https://docs.rs/scatter-net
[22] https://github.com/prokopschield/scatter-net

Chapter 7

Testing and Evaluation

7.1 Introduction

This chapter presents the testing and evaluation undertaken for the ScatterNet implementation. The primary objectives of this phase were to verify the functional correctness of the core components, assess the performance characteristics of data transfer operations, and conduct a qualitative evaluation of the system's security posture based on its design and the cryptographic primitives employed. Due to the scope of a Bachelor's thesis, exhaustive large-scale network testing and formal security audits were not performed; however, the conducted tests provide initial insights into the system's behavior and adherence to its design goals.

7.2 Functional Testing and Unit Tests

A suite of unit tests was developed to verify the correctness of individual modules within the ScatterNet implementation. These tests focused on the core library crates responsible for data processing and cryptographic operations, including:

- **`ps-hash`:** Verification of correct `Hash` generation according to the specified algorithm (SHA-256 $\oplus$ BLAKE3, `PackedInt` length, Reed-Solomon parity, and Base64 encoding).
- **`ps-pint16` and `ps-base64`:** Correctness of encoding and decoding for `PackedInt` and the custom Base64 scheme.
- **`ps-ecc`:** Validation of Reed-Solomon encoding and decoding capabilities for both the `Hash` parity and the Long ECC scheme's header and payload protection, including simulated error injection and correction.
- **`ps-datachunk` and `ps-cypher`:** Correct serialization, compression, AEAD encryption/decryption, and Long ECC application/removal for `DataChunk`s. This included verifying that data encrypted and then decrypted yielded the original plaintext and that authentication tags were correctly generated and verified.
- **`ps-hkey`:** Correct parsing and resolution of different `HKey` variants.

While these unit tests cover critical functionalities within each module, comprehensive coverage of all edge cases and complex interactions across the entire system remains an

area for future work. The successful execution of these unit tests provided a foundational level of confidence in the correctness of the core data manipulation and protection logic.

7.3 Performance Evaluation

A qualitative performance evaluation was conducted by testing the transfer of files of various sizes between two ScatterNet peer instances running on a local network. The primary focus was on the end-to-end throughput of the `PUT` and `FETCH` operations.

7.3.1 Methodology

Two peer instances, implemented using the `scatter-net` crate, were run on separate machines connected via a Gigabit Ethernet local area network. Files ranging from a few kilobytes to several megabytes were transferred using the `PUT` operation from a client peer to a storing peer, and subsequently retrieved using the `FETCH` operation. The time taken for these operations was observed.

7.3.2 Observations and Discussion

In the conducted tests, the transfer of files via ScatterNet's `PUT` and `FETCH` operations appeared to be primarily limited by the available network bandwidth between the two test nodes. Data transfers were observed to proceed rapidly, effectively saturating the network link capacity. This observation suggests that the overhead introduced by ScatterNet's data processing pipeline (serialization, compression, hashing, encryption, error correction) is not a significant bottleneck for point-to-point transfers under these local network conditions.

While these initial observations are positive, indicating efficient data handling, they do not constitute a rigorous quantitative performance benchmark. A more comprehensive performance analysis would require:

- Testing across wider area networks with varying latency and bandwidth characteristics.
- Measuring CPU and memory utilization on peers during data processing and transfer.
- Evaluating performance with a larger number of concurrent peers and operations.
- Profiling individual components (e.g., encryption, ECC encoding/decoding) to identify specific performance contributions.

Such extensive benchmarking was beyond the scope of this work but is recommended for future development to fully characterize the system's performance envelope and identify potential areas for optimization, especially for resource-constrained devices or large-scale deployments.

7.4 Security Evaluation

A formal security audit or penetration testing of the ScatterNet implementation was not conducted as part of this thesis. The security evaluation is therefore primarily based on a qualitative assessment of the design and the cryptographic primitives employed.

7.4.1 Design-Level Security

ScatterNet's security model relies on established cryptographic components and principles:

- **Confidentiality and Integrity:** Achieved through the use of the ChaCha20-Poly1305 AEAD scheme for encrypting `DataChunk` payloads. This provides strong, authenticated encryption, protecting data from unauthorized disclosure and modification. The security of this scheme is well-regarded in the cryptographic community [35].
- **Key Management:** The encryption key for a `DataChunk` is derived deterministically from the `SerializedDataChunk` itself (H(SerializedDataChunk)). This approach ensures that the key is intrinsically linked to the data it protects. The `HKey` then securely bundles the identifier for the encrypted data with this encryption key.
- **Data Integrity of Identifiers:** The `Hash` identifiers incorporate Reed-Solomon error correction, providing resilience against accidental corruption. The cryptographic hash functions (SHA-256 and BLAKE3) ensure the uniqueness and unforgeability of these identifiers under standard cryptographic assumptions.
- **Resilience to Data Corruption:** The Long ECC scheme applied to encrypted payloads provides an additional layer of protection against physical data corruption during storage or transit.
- **Participant Identification:** The use of Ed25519 public keys provides a strong cryptographic basis for peer identification, although the current protocol specification primarily uses these for addressing rather than explicit message authentication between peers.

7.4.2 Limitations and Future Work

While the design incorporates strong cryptographic primitives, the overall security of a deployed ScatterNet system would also depend on:

- **Secure Implementation:** Correct and secure implementation of all cryptographic operations and protocol logic is paramount. Vulnerabilities can arise from implementation errors even if the underlying cryptographic components are sound. The use of Rust and established cryptographic libraries aims to mitigate some of these risks.
- **Nonce Management:** The security of ChaCha20-Poly1305 critically depends on the uniqueness of nonces for each key. The deterministic derivation of the nonce from the encryption key's parity bits is a specific design choice that requires careful analysis to ensure sufficient entropy and non-collision properties in practice.
- **Transport Layer Security:** As ScatterNet defines a logical interface, the security of data in transit between peers (e.g., protection against active network attackers) relies on the security of the underlying transport mechanism (e.g., QUIC provided by iroh, or TLS if used with TCP).
- **Peer Discovery and Trust Management:** Mechanisms for secure peer discovery and establishing trust between peers are not part of the current core protocol specification and would need to be addressed in a complete system deployment.

- **Fair Use Policy Robustness:** The effectiveness of the reputation-based FUP against sophisticated or colluding malicious actors would require further analysis and potentially more advanced mechanisms in a production environment.

A comprehensive security audit by independent experts would be a necessary step before any production deployment of ScatterNet.

7.5 Conclusion

The testing and evaluation conducted for this thesis provide initial positive indications regarding the functional correctness and performance potential of the ScatterNet implementation. Unit tests confirmed the behavior of core data processing and cryptographic modules. Qualitative performance observations suggest that the protocol can efficiently utilize available network bandwidth for data transfers between two peers. The security of the system is founded on the use of strong, well-established cryptographic primitives and a design that emphasizes data confidentiality, integrity, and resilience.

However, the evaluation also highlights areas for future work. More extensive performance benchmarking across diverse network conditions and scales is required for a complete performance profile. Similarly, while the design incorporates robust security principles, a formal security audit and further analysis of specific aspects, such as nonce generation and the FUP's resilience, would be essential for a production-ready system. The current implementation serves as a solid foundation for these future endeavors.

Conclusion

This bachelor's thesis set out to address the challenges of secure data sharing in decentralized peer-to-peer (P2P) networks. Recognizing the limitations of existing solutions in providing comprehensive end-to-end security, the primary objective was to design and specify `ScatterNet`, a novel protocol tailored for such environments. The work successfully achieved this aim by delivering a multi-faceted solution encompassing a robust data serialization format, a secure encryption pipeline, a versatile data addressing mechanism, and a foundational communication interface.

The core contributions of this thesis are centered around the meticulous design of these components. A secure data serialization format was developed, starting with the `DataChunk` as the fundamental unit. This format incorporates a sophisticated, content-addressable `Hash` identifier, which combines the strengths of SHA-256 and BLAKE3, includes data length information via a custom `PackedInt` scheme, and is fortified with Reed-Solomon error correction parity for enhanced robustness. The data protection pipeline ensures confidentiality and integrity through Deflate compression followed by ChaCha20-Poly1305 authenticated encryption. The encryption key is deterministically derived from the data itself, and the nonce is carefully constructed from components of this key. Furthermore, a custom Long Error Correction Code (Long ECC) scheme, featuring a self-correcting header and iterative interleaved Reed-Solomon parity, provides an additional layer of resilience against physical data corruption for the encrypted payloads. Addressing and referencing these protected data units are managed by a flexible `HKey` system, supporting various data representations from directly embedded small data to complex, aggregated structures.

Building upon this secure data foundation, a minimalist peer-to-peer communication protocol was specified. Participants are identified by Ed25519 public keys, and the protocol defines core `FETCH` and `PUT` operations for data exchange. This interface is designed to be transport-agnostic, allowing for implementation over diverse network layers. To promote equitable resource usage, a local, subjective reputation system was introduced, enabling peers to manage interactions based on observed behavior.

The reference implementation of `ScatterNet`, realized as a collection of modular Rust crates, demonstrates the practical viability of the proposed design. Key architectural choices, such as the use of the iroh library for QUIC-based peer-to-peer communication and the Tokio asynchronous runtime, were made to leverage modern, efficient, and secure technologies. While the current implementation provides a solid foundation, areas such as automated peer discovery and advanced trust management mechanisms remain avenues for future development.

The largest contribution of this thesis, second only to the serialization format's design, is undoubtedly the array of Rust modules (crates) listen in Section 6.1. None of these are inherently tied to the `ScatterNet` project and therefore have the potential to be independently useful.

In conclusion, this thesis has presented a comprehensive design for the `ScatterNet` protocol, offering a significant step towards enabling truly secure, resilient, and user-centric data sharing in decentralized peer-to-peer networks. The detailed specification of its data formats and communication interface, coupled with the insights from its initial implementation, provides a valuable contribution to the field and a robust platform for future research and extension.

Bibliography

[1] *Fscrypt: Filesystem encryption for ext4 and f2fs* online. The Linux Foundation, 2023. Available at: https://www.kernel.org/doc/html/latest/filesystems/fscrypt.html. [cit. 2025-05-10].

[2] Anderson, R.; Biham, E. and Knudsen, L. *Serpent: A Proposal for the Advanced Encryption Standard (AES)*. AES Proposal Submission. National Institute of Standards and Technology (NIST), June 1998. Available at: https://www.cl.cam.ac.uk/archive/rja14/Papers/serpent.pdf.

[3] Bellare, M. and Namprempre, C. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology*. Springer-Verlag, October 2008, vol. 21, no. 4, p. 469–491. Available at: https://link.springer.com/article/10.1007/s00145-008-9026-x.

[4] Benet, J. IPFS - Content Addressed, Versioned, P2P File System. *ArXiv preprint arXiv:1407.3561*, July 2014. Available at: https://arxiv.org/abs/1407.3561.

[5] Berners Lee, T.; Fielding, R. T. and Masinter, L. M. *Uniform Resource Identifier (URI): Generic Syntax* RFC 3986. RFC Editor, january 2005. Available at: https://doi.org/10.17487/RFC3986.

[6] Bernstein, D. J. The Poly1305-AES message-authentication code. In: *Fast Software Encryption: 12th International Workshop, FSE 2005*. Springer, 2005, vol. 3557, p. 32–49. Lecture Notes in Computer Science. Available at: https://cr.yp.to/mac/poly1305-20050329.pdf.

[7] Bernstein, D. J. ChaCha, a variant of Salsa20. In: *Workshop Record of SASC 2008: The State of the Art of Stream Ciphers*. Lausanne, Switzerland: [b.n.], February 2008, p. 3–5. Available at: https://cr.yp.to/chacha/chacha-20080128.pdf.

[8] Bernstein, D. J. and Lange, T. *EBACS: ECRYPT Benchmarking of Cryptographic Systems* online. 2008–2025. Available at: https://bench.cr.yp.to/.

[9] Bonwick, J.; Ahrens, M.; Henson, V.; Maybee, M. and Shellenbaum, M. The Zettabyte File System. In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, p. 215–228. Available at: https://www.cs.hmc.edu/~rhodes/cs134/readings/The%20Zettabyte%20File%20System.pdf.

[10] Callas, J.; Donnerhacke, L.; Finney, H.; Shaw, D. and Thayer, R. *RFC 4880: OpenPGP message format*. RFC Editor, November 2007. Available at: https://www.rfc-editor.org/rfc/rfc4880.txt.

[11] Cohen, B. Incentives build robustness in BitTorrent. In: Berkeley, CA, USA. *Workshop on Economics of Peer-to-Peer Systems*. 2003, vol. 6, p. 68–72.

[12] Daemen, J. and Rijmen, V. *AES proposal: Rijndael*. AES Algorithm Submission. National Institute of Standards and Technology, 1999. Available at: https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf.

[13] Daemen, J. and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Berlin, Heidelberg: Springer-Verlag, 2002. Information Security and Cryptography. ISBN 978-3-642-07355-7.

[14] Deutsch, P. *DEFLATE Compressed Data Format Specification version 1.3*. RFC Editor, May 1996. Available at: https://www.rfc-editor.org/info/rfc1951.

[15] Dworkin, M. J. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D. National Institute of Standards and Technology, November 2007. Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf.

[16] Ferguson, N.; Schneier, B. and Kohno, T. *Cryptography Engineering: Design Principles and Practical Applications*. John Wiley & Sons, 2010. ISBN 978-0-470-47424-2.

[17] Fruhwirth, C. *LUKS On-Disk Format Specification* online. The LUKS Project, 2018. Available at: https://www.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf. [cit. 2025-05-07].

[18] Green, M. and Smith, M. Developers Are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy*, 2011, vol. 9, no. 2, p. 40–46. Available at: https://doi.org/10.1109/MSP.2010.198.

[19] Gueron, S. *Intel® Advanced Encryption Standard Instructions (AES-NI)*. White Paper. Intel Corporation, March 2010. Available at: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[20] Gueron, S. and Kounavis, M. E. *Intel® carry-less multiplication instruction and its usage for computing the GCM mode*. White Paper. Intel Corporation, May 2010. 10 p. Available at: https://cdrdv2-public.intel.com/836172/clmul-wp-rev-2-02-2014-04-20.pdf.

[21] Halcrow, M. A. ECryptfs: An enterprise-class encrypted filesystem for Linux. In: *Proceedings of the 2005 Linux Symposium*. 2005, vol. 1, p. 201–218. Available at: https://www.kernel.org/doc/ols/2005/ols2005v1-pages-209-226.pdf.

[22] IDRIX. *VeraCrypt Documentation* online. The VeraCrypt Project, 2023. Available at: https://www.veracrypt.fr/en/Documentation.html. [cit. 2025-05-10].

[23] Iyengar, J. and Thomson, M. *QUIC: A UDP-Based Multiplexed and Secure Transport* RFC 9000. RFC Editor, may 2021. Available at: https://doi.org/10.17487/rfc9000.

[24] Josefsson, S. and Liusvaara, I. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. RFC Editor, January 2017. Available at: https://www.rfc-editor.org/info/rfc8032.

[25] Josefsson, S. *The Base16, Base32, and Base64 Data Encodings* RFC 4648. RFC Editor, october 2006. Available at: https://doi.org/10.17487/RFC4648.

[26] Katz, J. and Lindell, Y. *Introduction to Modern Cryptography*. 3rdth ed. CRC Press, 2021.

[27] Kocher, P.; Jaffe, J. and Jun, B. Differential Power Analysis. In: Wiener, M., ed. *Advances in Cryptology — CRYPTO' 99*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, p. 388–397. ISBN 978-3-540-48405-9. Available at: https://link.springer.com/chapter/10.1007/3-540-48405-1_25.

[28] Krawczyk, H. The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In: Kilian, J., ed. *Advances in Cryptology — CRYPTO 2001*. Springer, 2001, vol. 2139, p. 310–331. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/3-540-44647-8_19.

[29] Langley, A.; Chang, W.-T.; Mavrogiannopoulos, N.; Strombergson, J. and Josefsson, S. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)* RFC 7905. RFC Editor, June 2016. Available at: https://doi.org/10.17487/RFC7905.

[30] Lin, S. and Costello, D. J. *Error Control Coding: Fundamentals and Applications*. 2ndth ed. Prentice Hall, 2004. ISBN 978-0130426727.

[31] McGrew, D. A. and Viega, J. The Galois/counter mode of operation (GCM). *Submission to NIST Modes of Operation Process*. Citeseer, 2004, vol. 20, p. 0278–0070.

[32] National Institute of Standards and Technology. *Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)* Federal Register. September 1997. Available at: https://www.govinfo.gov/content/pkg/FR-1997-09-12/pdf/97-24214.pdf.

[33] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. FIPS PUB 197. U.S. Department of Commerce, November 2001. Available at: https://doi.org/10.6028/NIST.FIPS.197-upd1. Includes Update 1 (May 29, 2023).

[34] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication FIPS PUB 180-4. Gaithersburg, MD: National Institute of Standards and Technology, august 2015. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[35] Nir, Y. and Langley, A. *ChaCha20 and Poly1305 for IETF Protocols* RFC 8439. RFC Editor, jun 2018. Available at: https://doi.org/10.17487/RFC8439.

[36] Oram, A. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA, USA: O'Reilly Media, 2001. ISBN 978-0-596-00110-0.

[37] Paar, C. and Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag Berlin Heidelberg, 2010. ISBN 978-3-642-04100-6.

[38] Reed, I. S. and Solomon, G. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*. SIAM, 1960, vol. 8, no. 2, p. 300–304.

[39] Rogaway, P. Authenticated-Encryption with Associated-Data. In: ACM. *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 2002, p. 98–107. Available at: https://doi.org/10.1145/586110.586125.

[40] Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 20th Anniversary Editionth ed. Hoboken, NJ, USA: Wiley, 2017. ISBN 978-1-119-43902-8.

[41] Souppaya, M. and Scarfone, K. *Guide to Storage Encryption Technologies for End User Devices*. Special Publication SP 800-111. National Institute of Standards and Technology (NIST), November 2007. Available at: https://csrc.nist.gov/publications/detail/sp/800-111/final. [cit. 2025-05-12].

[42] Stallings, W. *Cryptography and Network Security: Principles and Practice*. 7thth ed. Harlow, England: Pearson Education Limited, 2017. ISBN 978-1-292-15858-9.

[43] Tanenbaum, A. S. and Steen, M. van. *Distributed Systems*. 4thth ed. Maarten van Steen, 2023. ISBN 978-90-815406-4-3. Available at: https://www.distributed-systems.net.