

Sparse and Event-Based Client Designs for EVM-Compatible Blockchains

A Sparse Node Implementation for Ethereum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Philipp Slowak, BSc.

Matrikelnummer 01427655

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr. sc. ETH Georgia Avarikioti

Mitwirkung: Giulia Scaffino, MSc.

Wien, 1. Oktober 2025

Philipp Slowak

Georgia Avarikioti

Sparse and Event-Based Client Designs for EVM-Compatible Blockchains

A Sparse Node Implementation for Ethereum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Philipp Slowak, BSc.

Registration Number 01427655

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr. sc. ETH Georgia Avarikioti

Assistance: Giulia Scaffino, MSc.

Vienna, October 1, 2025

Philipp Slowak

Georgia Avarikioti



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Philipp Slowak, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Oktober 2025

Philipp Slowak



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Die Fertigstellung dieser Arbeit markiert den Abschluss eines wichtigen Kapitels in meinem Leben und wäre ohne die Unterstützung und Begleitung vieler Menschen, denen ich zu großem Dank verpflichtet bin, nicht möglich gewesen.

An erster Stelle möchte ich meiner Betreuerin Georgia (Zeta) Avarikioti danken. Sie hat mir die Möglichkeit gegeben, dieses Projekt zu verfolgen, und mich während des gesamten Programms mit Rat und Tat begleitet.

Mein tiefster und aufrichtigster Dank gilt jedoch Giulia Scaffino. Ihre unermüdliche fachliche Unterstützung, ihre wertvollen Anregungen und ihr unerschütterlicher Zuspruch waren in jeder Phase dieser Arbeit von entscheidender Bedeutung. Als meine wichtigste Ansprechpartnerin hat sie unzählige Stunden damit verbracht, Ideen zu diskutieren, Entwürfe zu lesen und kritisches Feedback zu geben, das diese Arbeit auf ein höheres Niveau gehoben hat. Ihre Erfahrung und Geduld haben diese Arbeit wesentlich bereichert, und ich bin zutiefst dankbar, so viel von ihr gelernt zu haben.

Ebenso danke ich für die bereichernden und inspirierenden Gespräche mit Jakov Mitrovski (Common Prefix), Thomas Thiery (Ethereum Foundation), Noah Citron (a16z crypto) und Matteo Maffei (TU Wien), deren Perspektiven von unschätzbarem Wert waren.

Persönlich möchte ich meiner Familie meinen größten Dank aussprechen – für ihre bedingungslose Liebe, ihren unerschütterlichen Glauben an mich und ihre unendliche Geduld während dieses langen Prozesses.

Meiner Freundin danke ich für ihre unermüdliche Unterstützung, ihr Verständnis und dafür, dass sie mir in dieser intensiven Zeit stets Freude und Ausgleich geschenkt hat. Dieser Erfolg wäre ohne dich nicht derselbe.

Schließlich danke ich auch meinen Freundinnen und Freunden für die willkommene Ablenkung, das gemeinsame Lachen und dafür, dass sie mich immer wieder daran erinnern haben, dass es auch ein Leben außerhalb der Wissenschaft und Universität gibt. Dieser Erfolg gehört auch euch.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

The completion of this thesis marks the end of an important chapter of my life and would not have been possible without the support and guidance of many individuals to whom I owe a great debt of gratitude.

Foremost, I wish to thank my supervisor, Georgia (Zeta) Avarikioti, for providing me with the opportunity to pursue this research and for her guidance throughout the program.

My most profound and sincere gratitude, however, is reserved for Giulia Scaffino. She provided invaluable guidance, insightful feedback, and unwavering encouragement that were instrumental at every stage of this research. As my primary point of contact, she spent countless hours discussing ideas, reviewing drafts, and providing the critical feedback that pushed this work to a higher standard. Her experience and patience greatly enriched this work, and I am profoundly grateful for the opportunity to have learned under her mentorship.

I am also thankful for the fruitful and inspiring discussions I had with Jakov Mitrovski (Common Prefix), Thomas Thiery (Ethereum Foundation), Noah Citron (a16z crypto), and Matteo Maffei (TU Wien). Their perspectives were invaluable.

On a personal note, I wish to express my immense gratitude to my family for their unconditional love, steadfast belief in me, and endless patience throughout this long process. You were my anchor and my motivation.

To my girlfriend, thank you for your unwavering support, understanding, and for being my constant source of joy and balance during this intense period. This achievement would not have been the same without you by my side.

Finally, to my friends, thank you for the distractions, the laughter, and for reminding me that there is a world outside of academia. This achievement is shared with all of you.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Moderne Blockchains verarbeiten mittlerweile Zehntausende Transaktionen pro Sekunde. Mit steigendem Durchsatz wachsen jedoch auch die Anforderungen für die Verifikation von Blockchains. Zentralisierte Node-as-a-Service (NaaS)-Anbieter (z.B. Infura oder Alchemy) bieten zwar praktische APIs, schaffen jedoch zusätzliche Vertrauensabhängigkeiten und bergen Risiken in Bezug auf Datenschutz und Zensurfreiheit. Ein selbst betriebener Full Node ermöglicht Datenzugang ohne zusätzliche Vertrauensannahmen, ist für die meisten Nutzerinnen und Nutzer jedoch aufgrund des hohen Ressourcenbedarfs kaum praktikabel. Im Gegensatz dazu arbeiten Light Clients deutlich ressourcenschonender, können dafür den vollständigen Anwendungszustand nicht rekonstruieren. Ein neuer Ansatz, der als *Sparse Client* (bzw. *Partially Stateless Client*) bekannt ist, ermöglicht dagegen die verifizierbare Überwachung eines Teilzustands der Blockchain, indem ausschließlich jene Transaktionen heruntergeladen, ausgeführt und gespeichert werden, die diesen Teilzustand lesen oder verändern. Bisher fehlt eine fundierte wissenschaftliche Aufarbeitung: Die einzige verfügbare Arbeit zu diesem Thema weist deutliche Limitierungen auf und wurde weder implementiert noch umfassend evaluiert.

In dieser Arbeit präsentieren wir zwei Sparse-Client-Protokolle für EVM-kompatible Blockchains: *Sparseth* für zustandsbasierte Synchronisation und *Eventeth* für ereignisbasierte Synchronisation. Beide Protokolle ermöglichen es Nutzerinnen und Nutzern, überprüfbare Teilmengen der globalen Transaktions- oder Ereignissequenz und des damit verbundenen Zustands zu verwalten, ohne dass zusätzlicher Validator-Aufwand erforderlich ist. Sparseth nutzt einen Interaktionszähler, um sicherzustellen, dass keine relevanten Transaktionen ausgelassen werden, während Eventeth eine kryptographische Hash-Kette einsetzt, um die Integrität und Vollständigkeit der Ereignisse zu gewährleisten. Im Gegensatz zu bestehenden Ansätzen arbeiten beide Protokolle vollständig auf der Ausführungsschicht und sind mit EVM-basierten Blockchains kompatibel.

Unsere formale Analyse zeigt, dass beide Protokolle im angenommenen Widersacher-Modell Sicherheit, Liveness und spärliche Gültigkeit garantieren. Unsere Implementierung in Go demonstriert die praktische Umsetzbarkeit: Event Nodes senken den Bandbreitenbedarf um über 95%, Sparse Nodes reduzieren die auszuführenden Transaktionen um 92% gegenüber Full Nodes. Die Gas-Kosten steigen um 4–16% für typische dApp-Transaktionen, ein Mehraufwand, der sich durch L2-Lösungen und ökonomische Anreize weiter mindern lässt.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Modern blockchains are scaling to tens of thousands of Transactions Per Second (TPS), but higher throughput increases resources needed for verification and state access. Centralized Node-as-a-Service (NaaS) providers (e.g., Infura, Alchemy) offer convenient APIs but introduce trust, privacy, and censorship risks. Running a full node is trustless but impractical for most users due to resource requirements, while light clients are efficient but cannot reconstruct application state. A new kind of blockchain client has recently been introduced under the name of *sparse client* (aka. *partially stateless client*): it verifiably monitors a substate of the ledger by only downloading, executing, and storing transactions that read from or write to the substate. As of today, this client is understudied and the only existing design has significant drawbacks and lacks implementation and evaluation.

This thesis introduces two sparse client protocols for EVM-compatible blockchains: *Sparseth* for state-based synchronization and *Eventeth* for event-based synchronization. Both enable clients to maintain verifiable subsets of the global ledger and state without validator overhead. Sparseth uses an interaction counter to ensure no relevant transactions are omitted, while Eventeth uses a cryptographic hash chain to verify event log integrity and completeness. Contrary to the existing design, both protocols operate entirely at the execution layer and are finally compatible with EVM-based chains.

We provide formal security analysis proving both protocols satisfy safety, liveness, and sparse validity under our adversarial model. Our implementation in Go demonstrates practical feasibility, achieving over 95% bandwidth reduction for event nodes and 92% computational reduction for sparse nodes compared to full nodes. Gas overhead analysis shows the protocols add 4–16% to typical dApp transaction costs, with mitigation strategies including L2 deployment and economic mechanisms.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
1.1 Contribution	3
2 Preliminaries	7
2.1 Notation	7
2.2 Cryptographic Primitives	7
2.3 Network Model	9
2.4 Atomic Broadcast and State Machine Replication	10
2.5 From Ledgers to Blockchains	12
2.6 Ethereum	13
3 Model	21
3.1 Cryptographic Assumptions	21
3.2 Network Assumptions	22
3.3 Ledger Model	22
3.4 Adversarial Model	23
3.5 Client Model	23
3.6 Sparse Node Model	24
3.7 Event Node Model	29
4 Sparse Node Protocol	33
4.1 Modes of Operation	34
4.2 Eventeth	35
4.3 Sparseth	41
4.4 Discussion	47
5 Analysis	51
5.1 Security Analysis of Eventeth	52
	xv

5.2	Security Analysis of Sparseth	53
5.3	Analysis of Sparse Node Resource Consumption	55
5.4	Data Availability Attacks	59
6	Implementation	63
6.1	Smart Contract Adaption for Sparse Monitoring	63
6.2	Proof-of-Concept Sparse Node Implementation	67
7	Evaluation	73
7.1	Gas Cost Overhead	73
7.2	Benchmarks	79
7.3	Operational Costs	88
8	Related Work	93
9	Conclusion	97
	Overview of Generative AI Tools Used	99
	List of Figures	101
	List of Tables	103
	List of Algorithms	105
	Acronyms	107
	Bibliography	111

Introduction

Modern blockchains are rapidly evolving toward ever-higher throughput. Emerging Layer-1 (L1) platforms like Solana, Sui, Aptos, and Monad are designed to achieve tens of thousands of Transactions Per Second (TPS), while Ethereum’s scaling roadmap targets comparable capacity [1]. While high throughput is desirable because it allows to on-board more users and developers, and it enables more complex applications, it comes at the cost of increased resource consumption, in terms of bandwidth consumption, computational overhead, and storage requirements for reading and verifying the entire chain. In other words, higher throughput comes with higher costs associated with verifying and reading the state of the chain.

Centralized Node-as-a-Service (NaaS) providers, such as MetaMasks’s Infura or Alchemy, offer convenient APIs to access blockchain data but introduce trust assumptions that undermine the values of blockchain systems. Moreover, managed nodes pose severe risk to user privacy and censorship resistance. Providers gain full visibility into user metadata, including IP addresses, request timing, and access patterns, which can reveal sensitive information about user behavior, such as trading strategies or application usage. Furthermore, reliance on a small number of dominant RPC providers creates a structural vulnerability to censorship. These entities are subject to external legal, political, and economic pressures and may be compelled to deplatform or censor specific users. Indeed, some of them already block access from entire countries¹. A stark example of this vulnerability is the sanctioning of the mixing service Tornado Cash by the U.S. Department of the Treasury’s Office of Foreign Assets Control (OFAC) in August 2022 [2]. Following this, major RPC providers, including Alchemy and Infura, complied with the sanctions by blocking access to the contracts’ data in their frontends. This action effectively prevented users of their services from interacting with the sanctioned smart contracts, demonstrating how centralized infrastructure can become a vector for enforcing regulatory decisions on-chain.

¹<https://support.metamask.io/start/can-i-use-metamask-in-my-country/>

The alternative—running a self-hosted full node—remains the most trustless, private, and censorship-resistant option for interacting with the chain. A full node achieves this by downloading, re-executing, and storing the entire ledger and state. However, its growing resource requirements render it impractical for most users with resource-constrained devices. On the other end of the spectrum, light clients offer a resource-efficient alternative by processing only block headers to identify the tip of the chain and verifying the inclusion of specific transactions. Yet this efficiency comes at a critical cost: light clients do not re-execute transactions and therefore cannot reconstruct the associated state. This fundamental limitation makes them unsuitable for operators of dApps and others who need near real-time data about smart contract and account states.

The lack of nodes with different trade-offs is especially problematic given that modern blockchains have evolved into general-purpose decentralized computers, hosting thousands of dApps, each with its own user base and economic incentives. In practice, an individual user or dApp operator is typically interested in only a small, specific subset of this global activity. Namely, the smart contract and events directly relevant to their operations. The existing client ecosystem forces a binary and unsatisfactory choice: either incur the prohibitive cost of a full node to verify everything, or settle for the limited capabilities of a light client that cannot verify application state.

Recognizing these challenges, there is active research on reducing the cost of blockchain verification. The Sunfish protocol [3], put forth by researchers of the Sui blockchain, represents a significant advance toward this goal by formalizing the concept of a *sparse node*: a client that maintains a verifiable subset of the global blockchain state. This approach drastically reduces resource requirements compared to a full node. However, Sunfish’s design has critical limitations: it requires validators to explicitly include additional commitments in block headers, thus burdening validators with an overhead that scales linearly with the number of supported substates, hindering practical deployment on live networks. More recently, Vitalik Buterin, co-founder of Ethereum, recently emphasized this priority in a post [4] on the Ethereum research forum, advocating a roadmap that explicitly favors designs enabling users to run nodes on consumer hardware. He emphasizes the concept of a *partially stateless client* that would verify blocks statelessly but maintain a small subset of the state, which it could serve RPC requests. While the post articulates the motivation and desired properties of such a client, it falls short of specifying a concrete protocol to realize it or defining any security guarantees that such a node would provide.

While both Sunfish and Ethereum’s scalability roadmap provide valuable conceptual guidance, there remains a clear need for a practical sparse node protocol that enables efficient verification without imposing an overhead on validators. This thesis addresses that gap by formalizing and implementing a new suite of protocols for EVM-compatible blockchains—including L1s like Ethereum, Avalanche, and Binance Smart Chain, as well as EVM-based Layer-2 (L2) solutions such as Arbitrum, Optimism, and Polygon—built entirely on execution-layer primitives.

1.1 Contribution

Our core contribution is the introduction of two novel protocols, each designed for a distinct synchronization paradigm. Our first protocol, named *Sparseth*, enables a client to selectively execute transactions, thereby maintaining a *sparse ledger*—a subsequence of the global transaction ledger—together with its associated *sparse state*. Our second protocol, named *Eventeth*, maintains a *sparse event log*—a sequence of specific events emitted by a contract. Eventeth design is inherently more lightweight, as it obviates transaction execution. Together, these protocols provide a new middle ground, addressing the limitations of existing full, light, and sparse clients. Building on the theoretical foundation first formalized by Sunfish [3], both of our proposed protocols make a fundamental design shift that prioritizes *practical deployability*.

The Sunfish sparse client is designed with broad compatibility in mind, targeting deployment across a wide range of blockchain platforms, including those without Turing-complete scripting capabilities (e.g., Bitcoin). Its approach is to require validators to generate a fresh commitment and embed it in block headers, enabling the Sunfish client to verifiably recompute the substate of interest. This aligns well with the philosophy of systems such as Sui, Solana, and Aptos, where validators take on the heavy lifting and clients remain as lightweight as possible but also pose a potential scalability bottleneck given the exponential number of possible substates. At the same time, another paradigm is emerging in blockchain architectures exemplified by projects like Monad, which decouples consensus from execution. Often described as *asynchronous execution*, in this model consensus and execution are pipelined, with execution occurring in a staggered manner after consensus completes: This moves execution out of the hot path of consensus, removing the time budget limit that other chains have for execution.

Our Sparseth and Eventeth clients put forth new sparse client designs that require only minimal adjustments—primarily to account for the shifted timing of commitment publication—to be fully compatible with the delayed execution paradigm and that remove the undesired linear overhead in the number of substates that Sunfish puts on validators. Specifically, our protocols are designed for EVM-compatible chains and require no changes to the underlying blockchain protocol, making both Sparseth and Eventeth ready for integration into today’s networks. We conjecture that they generalize to other smart contract platforms with similar capabilities (e.g., Solana, Sui, Aptos). We highlight that future Ethereum upgrades [5, 6] will introduce changes that further improve our nodes’ performances. Since we move the substate verification logic to the execution layer, Sparseth and Eventeth charge minimal gas cost on users that interact with sparse-node-compatible smart contracts (with zero overhead for EOAs). We discuss mitigation strategies for this overhead in Chapter 7.

To summarize, the main contributions of this work are:

- **Sparseth Design:** A protocol for state-based clients that enables the secure synchronization of a subset of the global ledger and associated state. A sparse

node selectively downloads, verifies, and re-executes transactions relevant to its monitoring scope to maintain a sparse ledger and corresponding sparse state. Sparseth is the first sparse node protocol that achieves practical deployability on existing EVM-compatible networks by requiring no changes to the underlying blockchain protocol. Furthermore, it introduces zero overhead for validators and aligns with asynchronous execution models, thereby enabling efficient and verifiable state synchronization.

- **Eventeth Design:** A protocol for event-based clients that provides a cryptographically verifiable log of on-chain events matching a client-specific monitoring scope. An event node selectively retrieves and verifies events, producing a sparse event log that serves as an authenticated, application-specific feed. Eventeth inherits the same practical advantages as Sparseth, providing the first immediately deployable solution for verifiable event streaming without blockchain protocol changes.
- **Security:** A formal security model for sparse and event nodes, defining the properties of *safety*, *liveness*, and *sparse validity*. Crucially, we provide formal proofs that both Sparseth and Eventeth satisfy these properties under our stated adversarial model.
- **Implementation and Evaluation:** A functional PoC implementation of Sparseth and Eventeth written in the Go programming language and leveraging the go-ethereum library, demonstrating the practical feasibility of our approach and providing the first concrete benchmarks for sparse node performance. The PoC implementation is publicly available at <https://github.com/pslowak/sparseth>.

The remainder of the thesis is organized as follows:

- **Chapter 2: Preliminaries** provides the necessary background on blockchain fundamentals, cryptographic primitives, and network models that underpin the subsequent protocol designs and analyses.
- **Chapter 3: Model** formalizes the foundational models and assumptions of the sparse node protocol, including cryptographic, network, ledger, adversarial, client, sparse node, and event node models, establishing the theoretical basis for our work.
- **Chapter 4: Sparse Node Protocol** details the design and specification of the novel Eventeth and Sparseth protocols, illustrating their mechanisms for efficient and verifiable event-based and state-based synchronization, respectively.
- **Chapter 5: Analysis** rigorously analyzes the security properties of Eventeth and Sparseth, formally proving their consistency and liveness guarantees. The chapter also provides a theoretical analysis of resource consumption, and concludes by examining resilience against Data Availability attacks.

- **Chapter 6: Implementation** describes the practical realization of the protocols, detailing architectural choices, technical challenges, and solutions employed in building the Eventeth and Sparseth client implementations.
- **Chapter 7: Evaluation** presents the empirical evaluation of the implemented protocols, assessing their resource efficiency, including an analysis of the gas cost overhead incurred by users as well as mitigation strategies for them.
- **Chapter 8: Related Work** surveys existing research and solutions in client design, light clients, and stateless verification, positioning our contributions within the broader academic and industrial landscape.
- **Chapter 9: Conclusion** summarizes the main findings of the thesis, reiterates the significance of our contributions, and discusses potential directions for future research and development.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Preliminaries

2.1 Notation

We use the following notation conventions throughout this work. Curly braces $\{\cdot\}$ denote sets, while parenthesis $\langle\cdot\rangle$ represent ordered sequences. For two sequences A and B , we write $A \preceq B$ to indicate that A is a (possibly empty) prefix of B , and $A \prec B$ if A is a strict prefix of B . Furthermore, we write $A \sim B$ to express that A and B are consistent, meaning that A is a prefix of B , or B is a prefix of A ($A \preceq B \vee B \preceq A$). For a set or sequence A , we write $|A|$ for the number of elements in A .

2.2 Cryptographic Primitives

In this section, we review the three primary cryptographic tools widely used in the blockchain ecosystem: digital signature schemes, cryptographic hash functions, and cryptographic commitments. For a comprehensive and formal treatment of these concepts, the reader is referred to two standard references [7, 8].

2.2.1 Digital Signature Scheme

A digital signature σ associates a message m with some sender S [7]. A digital signature scheme is formally defined by the following polynomial-time algorithms [8]:

- A probabilistic key-generation algorithm $\text{Gen} : 1^n \rightarrow (pk, sk)$ that takes as input a security parameter 1^n and outputs a pair of keys (pk, sk) , where pk denotes the public key and sk denotes the secret key (also referred to as the private key). We write this as $\text{Gen}(1^n)$.

- A signing algorithm $\text{Sign} : (sk, m) \rightarrow \sigma$ that, given a secret key sk and a message m drawn from some message space \mathcal{M} , produces a signature σ . We write this as $\text{Sign}_{sk}(m) \rightarrow \sigma$.
- A deterministic verification algorithm $\text{Verify} : (pk, m, \sigma) \rightarrow \{\top, \perp\}$ that takes as input a public key pk , a message $m \in \mathcal{M}$, and a signature σ , and outputs \top if the signature is valid for the message under the given public key, and \perp otherwise. We write this as $\text{Verify}_{pk}(m, \sigma)$. We require that $\text{Verify}_{pk}(m, \text{Sign}_{sk}(m)) = \top$ holds for every message $m \in \mathcal{M}$ (correctness).

Using the algorithms defined above, a sender S first runs $\text{Gen}(1^n)$ to generate a key pair (pk, sk) . The public key pk is then made publicly accessible as belonging to S , enabling any party to obtain a legitimate copy. To authenticate a message m , the sender computes a signature σ using the signing algorithm and sends (m, σ) . Upon receiving this pair, a receiver in possession of pk can verify the authenticity of m by evaluating $\text{Verify}_{pk}(m, \sigma)$. If the verification algorithm outputs \top , this confirms both that the message originated from S and that it has not been tampered with in transit [8].

2.2.2 Cryptographic Hash Functions

A cryptographic hash function $h : D \rightarrow R$ is a function that maps an input from a domain D , consisting of strings of arbitrary finite length, to a fixed-length output in the range R , referred to as digest. The fact that variable-length inputs are mapped to fixed-length outputs is known as *compression* [7].

In addition to *compression* and *ease of computation* (i.e., given h and an input x , $h(x)$ is computable in polynomial-time), we require cryptographic hash functions to satisfy the following properties [7]:

- **Preimage Resistance:** For any digest y it is computationally infeasible to find a preimage x , such that $h(x) = y$.
- **Second Preimage Resistance:** For any specified input x it is computationally infeasible to find a second preimage x' with $x \neq x'$, such that $h(x) = h(x')$.
- **Collision Resistance:** It is computationally infeasible to find any two distinct inputs x and x' which have the same digest, i.e., $h(x) = h(x')$.

The properties above are listed in order of increasing strength. In particular, collision resistance implies second preimage resistance, which in turn implies preimage resistance. For a formal discussion of these implications, the reader is referred to [8].

2.2.3 Commitment Schemes

A commitment scheme is a fundamental cryptographic primitive built upon cryptographic hash functions introduced in Section 2.2.2. It allows one party (the *committer*) to commit to a chosen message m by producing a short commitment value c , which can be published or shared. Later, the committer can reveal the original message m , allowing any verifier to check that c corresponds to m .

A secure commitment scheme must satisfy two essential properties:

- **Hiding:** The commitment reveals no information about the committed message.
- **Binding:** It is computationally infeasible for the committer to find two distinct messages m, m' with $m \neq m'$ that produce the same commitment c .

2.3 Network Model

The feasibility, correctness, and performance of distributed consensus protocols are profoundly influenced by the underlying assumptions about time and message delivery within the network. Different models capture these assumptions along a spectrum, ranging from the strong guarantees of *synchronous* networks, to the minimal constraints of *asynchronous* systems. Between these extremes lies the *partially synchronous* network model, which reflects the more realistic operating conditions of many practical distributed systems, including blockchain networks.

In this section, we outline the synchronous, asynchronous, and partially synchronous models, as they provide the foundation for the design and analysis of the protocols discussed in subsequent sections.

2.3.1 Synchronous Model

Time proceeds in discrete rounds. In this setting, every message initiated by an honest node is delivered within at most $\Delta > 0$ time units, thereby ensuring synchrony in communication. Additionally, nodes share a global notion of time formalized as a fixed upper bound $\phi > 0$ on the relative time drift between the local clocks of any two nodes, thereby maintaining synchrony of nodes. Because both Δ and ϕ are public parameters—known to any node a priori—these parameters can be leveraged in the design and analysis of distributed protocols [9].

2.3.2 Asynchronous Model

Unlike the synchronous model, the asynchronous model makes no assumptions about timing. There is no deterministic upper bound Δ on message transmission time. Messages may be delayed arbitrarily long, arrive out of order, or be interleaved in any fashion, although any message sent by an honest node is eventually delivered. Furthermore, nodes

have no access to a shared notion of time, and the rate ϕ at which their local clocks advance is unconstrained in the asynchronous model [10, 9].

2.3.3 Partially Synchronous Model

The partially synchronous model provides a more realistic abstraction of real-world network conditions than the idealized completely synchronous or completely asynchronous models. It embodies the practical observation that distributed systems typically operate synchronously in normal conditions, but may temporarily deviate from synchrony due to network disruptions or extreme events [11].

In a partially synchronous system, fixed upper bounds Δ and ϕ on message delay and clock drift, respectively, are assumed to exist, but they are not guaranteed to hold from the outset. Instead, there is a finite but unknown Global Stabilization Time (GST), after which the system transitions to synchronous behavior. Prior to GST, the network may exhibit fully asynchronous behavior, with unbounded message delays and arbitrary clock drift. After GST, the system guarantees that every message sent between honest nodes is delivered within at most Δ time units. Formally, a message sent by every honest node at round r is guaranteed to be received by every honest node by round $\max\{r, \text{GST}\} + \Delta$ [12].

2.4 Atomic Broadcast and State Machine Replication

Byzantine Fault-Tolerant (BFT) protocols have been a central topic in distributed systems research for decades [13, 14], with applications ranging from fault-tolerant databases to mission-critical control systems. In recent years, the advent of blockchain technologies has renewed interest in these protocols, as they form the theoretical foundation for achieving consensus in open, adversarial, and large-scale environments.

2.4.1 Atomic Broadcast

The Byzantine Atomic Broadcast (BAB) problem extends the reliable broadcast primitive to guarantee a consistent global ordering of messages, even in the presence of Byzantine faults. Let $\Pi = \{p_1, p_2, \dots, p_n\}$ denote the set of $n = 3f + 1$ processes, among which at most f are Byzantine. A sender process $p_k \in \Pi$ can transmit a message m in round r using the operation

$$\text{send}_k(m, r), \quad (2.1)$$

and each process p_i produces an output

$$\text{deliver}_i(m, r, p_k), \quad (2.2)$$

indicating that $p_i \in \Pi$ irrevocably accepted m sent by p_k in round r . The BAB abstraction guarantees the following properties [15, 16]:

- **Validity:** If an honest process p_k calls $\text{send}_k(m, r)$, then every other honest process p_i outputs $\text{deliver}_i(m, r, p_k)$.
- **Agreement:** If an honest process p_i outputs $\text{deliver}_i(m, r, p_k)$, then every other honest process p_j eventually outputs $\text{deliver}_j(m, r, p_k)$.
- **Integrity:** For each round r and process $p_k \in \Pi$, an honest process p_i outputs $\text{deliver}_i(m, r, p_k)$ at most once.
- **Total Order:** For any two messages m and m' , all honest processes agree on their relative delivery order. That is, if an honest process p_i outputs $\text{deliver}_i(m, r, p_k)$ before $\text{deliver}_i(m', r', p'_k)$, then no other honest process p_j delivers m' before $\text{deliver}_j(m, r, p_k)$.

The first three properties—validity, agreement, and integrity—are inherited from the reliable broadcast abstraction. In the context of blockchain systems, the BAB abstraction serves as a primitive for the separation of transaction sequencing and execution as done in [17] or [18]. It provides a mechanism to propose a totally ordered sequence of transactions, which can then be applied consistently across all replicas of a state machine.

2.4.2 State Machine Replication

State Machine Replication (SMR) is a general approach for implementing fault-tolerant services in distributed systems. Historically, SMR was employed to ensure consistency and liveness in replicated databases and other critical services. The core idea is to maintain multiple replicas of a service, each modeled as a deterministic state machine. Each replica receives inputs from the environment, applies a state transition function to the current state, and produce the same outputs in a deterministic manner. The main challenge is to ensure that these replicas process the same sequence of inputs in the same order, thereby remaining synchronized despite faults [19].

Blockchain systems can be viewed as a decentralized instantiation of the SMR paradigm. In this context, clients submit transactions to one or more participating nodes in the network, which collectively execute a consensus protocol to agree on a total ordering of these transactions. Each honest node then applies these transactions to its local state using a deterministic state transition function. As a result, all honest nodes maintain the same sequence of transactions in their local ledgers and update their local state deterministically, thereby ensuring a consistent global state across the network. Thus, honest nodes produce identical outputs when responding to requests.

Formally, let $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ denote the set of $n = 3f + 1$ nodes participating in the protocol, among which at most f are Byzantine. We denote $\mathcal{H} \subseteq \mathcal{P}$ the set of honest nodes, with cardinality $|\mathcal{H}| = 2f + 1$. Each node $p_i \in \mathcal{H}$ maintains a local log \mathcal{L}_i^r defined as the ordered sequence of values known to p_i after processing all inputs up to round r . For a value tx , the notation $\text{tx} \in \mathcal{L}_i^r$ indicates that tx is included in the local log of node p_i at the conclusion of round r .

A protocol implements SMR if it satisfies *consistency* and *liveness* [20, 21]

Definition 1 (SMR Consistency). A SMR protocol satisfies consistency if, at any given round, the local logs of all honest nodes are mutually consistent. Formally, for every two rounds r_1, r_2 , and for every two honest nodes $i, j \in \mathcal{H}$ the local log $\mathcal{L}_i^{r_1}$ is a prefix of the local log $\mathcal{L}_j^{r_2}$ or vice-versa. Formally:

$$\forall r_1, r_2 \forall i, j \in \mathcal{H} : \mathcal{L}_i^{r_1} \sim \mathcal{L}_j^{r_2}. \quad (2.3)$$

Consistency is a *safety* property that guarantees the system never reaches a state in which two honest nodes disagree on the relative ordering of inputs. Although the local logs of individual nodes may temporarily lag behind others, these logs are always prefixes of the logs maintained by other honest nodes.

Definition 2 (SMR Strong Liveness). A SMR protocol satisfies strong liveness if every client input that is submitted to all honest nodes is eventually included in the local logs of every honest node. Formally, for any input tx proposed by a client to all honest nodes in \mathcal{H} by round r , there exists a round $r' \geq r$ such that for every $r'' \geq r'$ and for every honest node $p_k \in \mathcal{H}$, $\text{tx} \in \mathcal{L}_k^{r''}$.

Liveness, conversely, is a *progress* property that ensures every valid input proposed by a client is eventually reflected in the local logs of all honest nodes.

Definition 3 (SMR Security). A SMR protocol is secure if it satisfies consistency and strong liveness.

2.5 From Ledgers to Blockchains

While a ledger is an ordered sequence of transactions, the blockchain paradigm extends this abstraction with additional structure. In a blockchain, transactions are aggregated into blocks. These blocks are chained together using a cryptographic hash as a means of reference and fraud protection [22].

A block consists of a *header* and a *body*. While the block body encapsulates an ordered batch of transactions, the block header may include additional fields that enhance the capabilities of a ledger. Examples of such fields include data structures for light [22] and sparse [3] client verification, sources of pseudo-randomness for leader election [23], or mechanisms that support timeliness, a property where Unix timestamps recorded on-chain do not deviate arbitrarily from real-world time [24]. In PoW blockchains, with variable difficulty, these timestamps are particularly important because they are used in the difficulty adjustment algorithm, which regulates the network's block production rate [25]. Additionally, timestamps are also used by transactions to specify a *locktime*, enabling conditional execution based on a minimum timestamp [26].

Despite these extensions, a blockchain can be reduced to a ledger. Let \mathcal{B} denote a blockchain of height h , and \mathcal{L} denote the corresponding ledger. We define \mathcal{B} as a finite, ordered sequence of blocks

$$\mathcal{B} = \langle b_1, b_2, \dots, b_h \rangle, \quad (2.4)$$

where each block $b_i = \langle \mathbf{tx}_{i,1}, \mathbf{tx}_{i,2}, \dots, \mathbf{tx}_{i,m_i} \rangle$ is a finite, ordered sequence of m_i transactions. We define the ledger projection

$$\mathcal{P}(\mathcal{B}) = \langle \mathbf{tx}_{1,1}, \mathbf{tx}_{1,2}, \dots, \mathbf{tx}_{1,m_1}, \mathbf{tx}_{2,1}, \mathbf{tx}_{2,2}, \dots, \mathbf{tx}_{h,1} \dots \mathbf{tx}_{h,m_h} \rangle \quad (2.5)$$

as the concatenation of the transaction sequences contained in all blocks. The resulting ledger $\mathcal{L} = \mathcal{P}(\mathcal{B})$ is therefore an ordered sequence of all transactions in the blockchain, preserving their total order.

2.6 Ethereum

Ethereum is an open-source, permissionless blockchain system that introduced Turing-complete scripting capabilities, enabling the development of complex dApps without reliance on a centralized entity. Although the concept of *smart contracts* predates Ethereum [27], it was the first platform to realize the vision of a decentralized computer at scale [28, 29]. This section provides the technical foundations necessary for understanding Ethereum’s architecture, with a focus on the execution-layer components and client models most relevant to the protocols developed in this work.

2.6.1 Foundational Concepts

We model Ethereum as a transaction-based state machine. The system starts from a well-defined *genesis state* \mathcal{S}^0 . Given the current state \mathcal{S}^i and a *valid* transaction \mathbf{tx}_{i+1} , the next state \mathcal{S}^{i+1} is determined by a deterministic state transition function

$$\mathcal{S}^{i+1} = \delta(\mathcal{S}^i, \mathbf{tx}_{i+1}) = \delta(\dots \delta(\delta(\mathcal{S}^0, \mathbf{tx}_0), t_2) \dots \mathbf{tx}_{i+1}). \quad (2.6)$$

Ethereum is an account-based blockchain. The world state

$$\mathcal{S} : K \rightarrow V \quad (2.7)$$

is a key-value mapping with a domain K and range V . In Ethereum, $k \in K$ corresponds to the 160-bit address of an account, and $v \in V$ is the Recursive Length Prefix (RLP) encoded state of this account [29].

2.6.2 Ethereum Accounts

An Ethereum account is an entity associated with an Ether (ETH) balance and capable of sending messages within the network. Ethereum distinguishes between two types of accounts [28]:

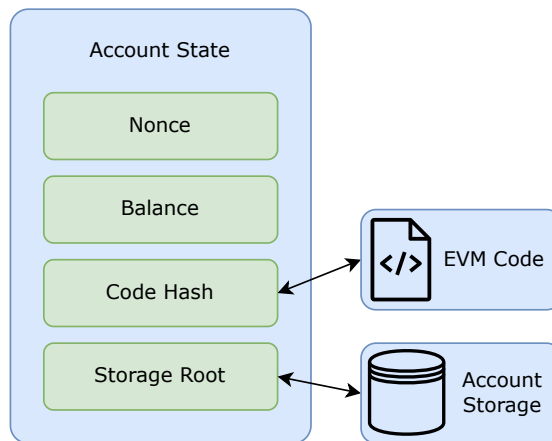


Figure 2.1: Illustration of an Ethereum account. For EOAs, the fields corresponding to the code hash and storage root are set to the empty hash, reflecting the absence of associated EVM code and account storage.

- **EOAs:** An Externally-Owned Account (EOA) consists of a cryptographic key pair, comprising a private key and a corresponding public key. Control over an EOA is granted to any party in possession of its private key. Only EOAs are capable of *initiating* transactions on the Ethereum network.
- **Contract accounts:** A contract account is represented by a smart contract deployed on the Ethereum network. Unlike EOAs, control of a contract account is governed by the logic encoded within its code rather than a private key. Deploying a smart contract incurs a cost, as it consumes storage resources on the network. Execution of contract code is triggered by a transaction, either from an EOA or contract account.

Ethereum accounts (both EOAs and contract accounts) can be fully characterized by the following four fields [29], illustrated in Figure 2.1:

- **Nonce:** The nonce is a scalar value indicating the total number of transactions sent from this address or, for accounts associated with code, the number of contract creations initiated by this account. Only one transaction with a given nonce can be executed per account, thereby preventing replay attacks in which a signed transaction is repeatedly broadcast and re-executed multiple times.
- **Balance:** The balance is a scalar value representing the amount of Wei held by this account, where 1 Ether = 10^{18} Wei.
- **Code:** The EVM bytecode to be executed whenever the account receives a message call. Unlike other account fields, the code is immutable once deployed. The code is stored in the state database under its corresponding hash for efficient retrieval. For EOAs, the code hash is the hash of the empty string.

- **Storage:** The storage of an account is a key-value mapping, encoded into a Merkle-Patricia Trie (MPT), which is empty by default. The root hash of the MPT is stored within the account state to provide a succinct commitment to the storage contents. For EOAs, this root hash always corresponds to that of the empty trie.

2.6.3 Gasper

In a decentralized network such as Ethereum, multiple parties—referred to as validators—can independently propose a new block on some earlier ancestor. To ensure that all participants converge on a single canonical chain, the network requires a consensus protocol. Since the Paris hard fork, consensus has been delegated to the Beacon Chain which implements a Proof of Stake (PoS) protocol known as Gasper. In a PoS protocol, the validators voting power is proportional to the amount of stake they have locked in the system [30].

The Gasper protocol combines Casper FFG [31] with LMD GHOST [32]. Casper Friendly Finality Gadget (FFG) marks certain blocks in a blockchain *finalized*, i.e., irrevocably part of the canonical chain. Casper works on top of a provided protocol. Latest Message Driven (LMD) Greedy Heaviest-Observed Sub-Tree (GHOST) is a fork-choice rule which defines how the head of the chain is selected at any given time. Because validators may propose blocks simultaneously, the blockchain can temporarily evolve into a tree rather than a strictly linear sequence. This implies that beyond a certain block, multiple states may co-exist. In LMD GHOST validators send attestations in support of specific blocks, and the fork-choice rule determines the most likely canonical chain [33].

Casper FFG Casper is an overlay protocol that operates on top of a block proposal mechanism. Casper is responsible for finalizing blocks, thereby determining a canonical chain that participants can rely on as irreversible. Casper provides *accountable safety*, while liveness depends on the underlying block proposal protocol. Accountable safety means that two conflicting checkpoints—checkpoints on different branches—cannot both be finalized unless a set of validators controlling a threshold of stake provably violates the protocol rules. If such a violation occurs, Casper FFG can identify the misbehaving validators, enabling the system to penalize them, addressing the *nothing at stake* problem [31, 33].

LMD GHOST LMD GHOST is a fork-choice rule that extends the heaviest observed chain. The weight of a chain is determined by the total stake of validators whose most recent attestations support blocks on that branch. By always following the branch with the greatest cumulative attested stake, LMD GHOST provides a deterministic method for resolving forks and ensuring convergence toward a single canonical chain [32, 33].

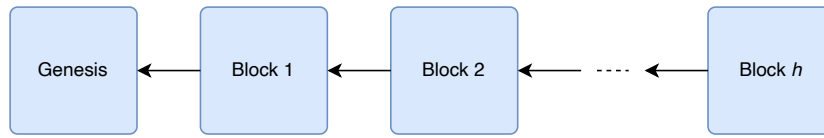


Figure 2.2: Illustration of a blockchain as a sequence of blocks, each containing a reference to the previous block via a cryptographic hash. The genesis block serves as the starting point and has no predecessor.

2.6.4 Data Structures

Blocks In Ethereum, ordered sequences of valid transactions are grouped into blocks. We denote the block at height h as

$$b_h = \langle \text{tx}_{h,0}, \text{tx}_{h,1}, \dots, \text{tx}_{h,m} \rangle. \quad (2.8)$$

Except for the genesis block, each block references its immediate predecessor via a cryptographic hash, thereby forming a chain of blocks, as illustrated in Figure 2.2. Formally, we can define the block-level state transition function as

$$\Delta(\mathcal{S}^i, b_{i+1}) = \delta(\dots \delta(\delta(\mathcal{S}^i, \text{tx}_{i+1,0}), \text{tx}_{i+1,1}), \dots, \text{tx}_{i+1,m}), \quad (2.9)$$

where Δ reduces a block to a single state transition by folding δ from Equation (2.6) over the transaction sequence it contains [29].

A block is a data structure consisting of a block header and a sequence of transactions, along with associated metadata. The execution of all transactions contained in a block transitions the global state from that of the previous block to a new state, as illustrated in Figure 2.4. The block header contains, among other fields, cryptographic commitments to three MPTs: the state trie, which encodes the world state; the transactions trie, which encodes the set of transactions in the block; and the recipients trie, which encodes the receipts resulting from transaction execution [29].

In the following, we consider a blockchain in which each block header includes, but is not limited to, three commitments that facilitate efficient client synchronization and verification:

- A commitment to the global state after block execution, C_S .
- A commitment to the transactions included in this block, C_T .
- A commitment to the outcomes of these transactions, referred to as transaction receipts, C_R .

The block structure is illustrated in Figure 2.3.

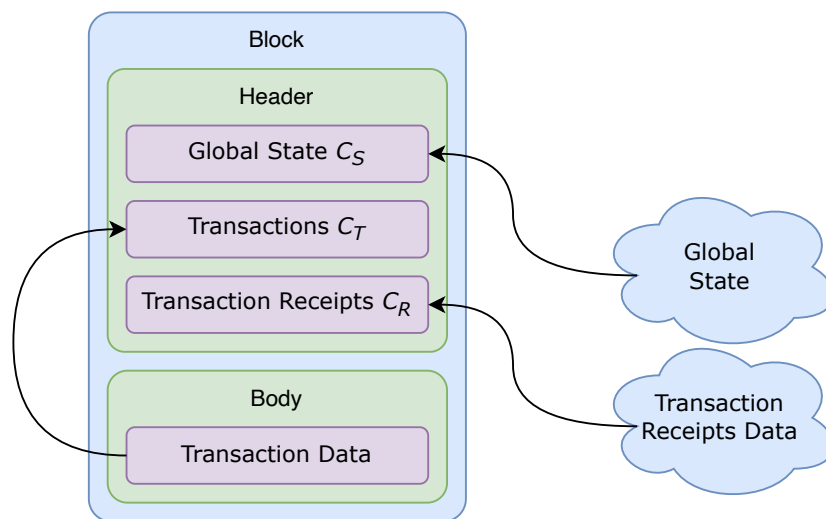


Figure 2.3: Illustration of an Ethereum block incorporating commitments that enable efficient client reads. The block is divided into a header, which contains metadata, and a body, which contains the sequence of transactions. In particular, the header includes three key commitments: C_S , the global state resulting from the execution of the block's transactions; C_T the transactions themselves, and C_R , the corresponding outcomes of these transactions (transaction receipts).

Modified Merkle Patricia Trie Ethereum encodes its state using a modified Merkle-Patricia Trie (MPT), a hybrid structure that combines properties of Merkle Trees with the prefix-compression features of PATRICIA tries [34]. This construction cryptographically links all data in the tree, resulting in a single deterministic root hash that serves as a commitment to the entire dataset [35]. The MPT enables efficient verification of state elements through Merkle proofs while logarithmic-time complexity for lookups, inserts and deletions.

In the execution layer of Ethereum, four distinct MPTs maintain important components of the blockchain state:

- **State Trie:** There is a single global state trie which is updated every time a block is executed. This trie is organized as a key-value store that maps account addresses to accounts. Its root hash corresponds to the state commitment C_S .
- **Storage Trie:** Each contract account has its own storage trie, which records all persistent contract data. The storage trie maps storage positions to stored values.
- **Transactions Trie:** The transactions trie contains all transactions included in a block. It maps the transaction index within the block to the corresponding transaction. Its root hash corresponds to the transaction commitment C_T .

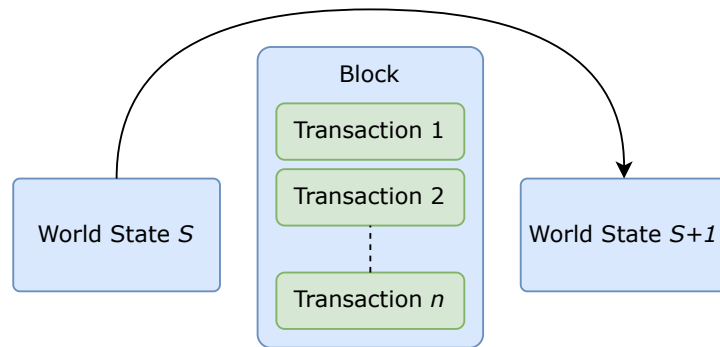


Figure 2.4: Illustration of an Ethereum state transition. The execution of all transactions in a block transforms the world state from the state root of the parent block to the new state root recorded in the block header.

- **Recipients Trie:** The receipts trie stores the transaction receipts for a block, mapping each transaction index to its associated receipt. Its root hash corresponds to the transaction receipt commitment C_R .

Transactions A transaction is a single cryptographically-signed instruction issued by an EOA to alter the state of the network. The sender of a transaction must be an EOA, as contract accounts cannot directly initiate transactions. Transactions fall into two categories: those that trigger message calls, and those that create new contract accounts, referred to as contract creation transactions [29].

With EIP-2718, transactions are encapsulated within a typed transactions envelope, enabling the definition of transaction types that support additional fields while preserving backward compatibility with legacy transaction formats [36].

2.6.5 Node Types

Today, the Ethereum Layer-1 (L1) ecosystem consists of two types of nodes: full nodes and light clients.

Full Nodes Full nodes download and verify the entire chain [37]. Since *The Merge* an Ethereum full node has to run two clients: a consensus client and an execution client. Figure 2.5 illustrates the architecture of an Ethereum full node. The consensus client implements the PoS protocol, while the execution client is responsible for receiving transactions, broadcasting them, executing them in the EVM, and maintaining the associated global state. Execution clients also expose data to external applications via the JSON-RPC API, and communicate with consensus clients through the Engine API. By default, full nodes synchronize the chain from the genesis block to the present head. To ensure correctness, they rely on at least one honest peer during synchronization [38].

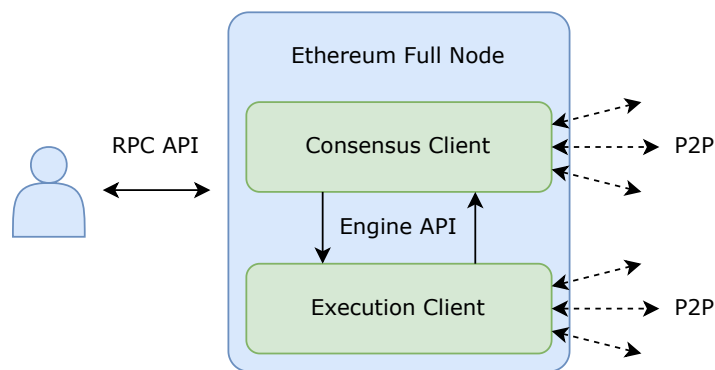


Figure 2.5: An Ethereum full node runs two clients: a consensus client and an execution client. Together, these components maintain the blockchain and enable users to interact with the Ethereum network.

Light Nodes Light clients provide a resource-efficient alternative to full nodes by avoiding the need to download and verify the entire blockchain. Instead, they synchronize with the chain by downloading only block headers and requesting cryptographic proofs from providers to verify the inclusion of specific transactions [37]. Such clients can vary in their lightness, but typically require substantially less resources than full nodes. In Ethereum, light clients rely on the *sync committee*, a set of 512 validators that rotates every 256 epochs. During its term, the sync committee collectively signs block headers, allowing light clients to verify the latest state without downloading all intermediate blocks. Since the committee changes every 256 epochs, the light client needs to keep track of the active committee. Thereto, the current committee includes a handover message in the block header that designates the next committee. Light clients operate under the assumption of an honest supermajority within the sync committee [38].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

Model

This chapter formalizes the foundational models and assumptions that underpin the design and analysis of the sparse node protocol. We begin by specifying the required cryptographic primitives, including digital signature schemes, cryptographic hash functions, and commitment schemes. These primitives are essential for ensuring the security and integrity of blockchain protocols.

Next, we establish the *network model*, which captures the communication assumptions between sparse nodes, full nodes, and potential adversaries. This model adopts a *partially synchronous* setting to reflect the operational conditions of practical blockchain systems.

We then formalize the *ledger model*, defining the structure of the ledger, its state transition function, and the representation of transactions and events. Building on this, we define the *adversarial model*, specifying the capabilities and limitations of an adversary attempting to violate security guarantees, including their control over network scheduling, block production, and data availability.

The chapter then introduces the *client model*, which serves as the foundation for sparse verification in our protocol. This allows us to precisely characterize the *sparse node model*, capturing its selective synchronization based on state predicates and formally defining its core security properties.

Finally, we extend the discussion to *event-based synchronization* by introducing the *event node model*. By providing these formal definitions, this chapter establishes the theoretical foundation for the protocol specifications and security proofs developed in subsequent chapters.

3.1 Cryptographic Assumptions

The security of blockchain protocols fundamentally relies on well-established cryptographic primitives. These primitives provide essential guarantees for core functionalities such as

authenticity, integrity, and resistance to forgery or tampering. In this section, we state the cryptographic assumptions that underpin our protocol.

We assume the digital signature scheme, formally introduced in Section 2.2.1, satisfies the standard notion of Existential Unforgeability Under Chosen Message Attack (EUF-CMA). This property ensures that no computationally efficient adversary can produce a valid signature on a new message, even after obtaining signatures on an arbitrary number of adaptively chosen messages.

Furthermore, we assume the existence of a secure cryptographic hash function, as described in Section 2.2.2. The hash function must be preimage resistant, second preimage resistant, and collision resistant, thereby preventing adversaries from finding collisions or inverting the function.

Finally, we rely on a secure commitment scheme, detailed in Section 2.2.3, which is both hiding and binding. These cryptographic assumptions form the basis for the security properties discussed in the subsequent sections.

3.2 Network Assumptions

The network model defines the communication environment in which our protocol operates. In this work, we assume that communication between the sparse client and the set of full nodes occurs in a *partially synchronous* network, as formally introduced in Section 2.3.3.

3.3 Ledger Model

The ledger model specifies the structure and evolution of the ledger maintained by the protocol. We model a ledger \mathcal{L} as the output of a BFT-SMR protocol as introduced in Section 2.4.2. Let

$$\mathcal{N} = \{1, 2, \dots, n\} \quad n = 3f + 1 \quad (3.1)$$

denote the set of nodes participating in the protocol, where at most f nodes are controlled by an adversary. We denote $\mathcal{H} \subseteq \mathcal{N}$ the set of honest nodes, with cardinality $|\mathcal{H}| = 2f + 1$.

Each node $i \in \mathcal{H}$ is modeled as a deterministic state machine maintaining a state \mathcal{S}^r at round r . The ledger state is represented as a mapping from a finite set of unique keys K to corresponding values on a domain V , i.e., $\mathcal{S} : K \rightarrow V$. Upon receiving a new transaction \mathbf{tx} from the environment, a node transitions from \mathcal{S}^r to \mathcal{S}^{r+1} by applying a deterministic state transition function δ

$$\mathcal{S}^{r+1} = \delta(\mathcal{S}^r, \mathbf{tx}). \quad (3.2)$$

Let $\mathcal{L}_i^r = (\mathbf{tx}_1, \mathbf{tx}_2, \dots, \mathbf{tx}_r)$ denote the ledger—a sequence strictly ordered of transactions—maintained by a node $i \in \mathcal{H}$ at round r , with the corresponding state

$$\mathcal{S}^r = \delta(\dots \delta(\delta(\mathcal{S}^0, \mathbf{tx}_1), \mathbf{tx}_2) \dots, \mathbf{tx}_r) \quad (3.3)$$

for some defined genesis state \mathcal{S}^0 . We write $\text{tx} \in \mathcal{L}_i^r$ if the transaction tx appears in the ledger of node $i \in \mathcal{H}$ at round r .

A ledger is said to be *secure* if it satisfies the following two properties: consistency and strong liveness. Consistency ensures that honest nodes never disagree on the order of transactions, while liveness guarantees that every valid transaction is eventually included in the ledger of every honest node.

Definition 4 (Ledger Consistency). A ledger satisfies consistency, if for every two rounds r_1, r_2 , and for every two honest nodes $i, j \in \mathcal{H}$ the ledger $\mathcal{L}_i^{r_1}$ is a prefix of the ledger $\mathcal{L}_j^{r_2}$ or vice-versa. Formally:

$$\forall r_1, r_2 \forall i, j \in \mathcal{H} : \mathcal{L}_i^{r_1} \sim \mathcal{L}_j^{r_2}. \quad (3.4)$$

In this definition, honest nodes may temporarily lag behind others, but can never disagree on the relative ordering of transactions. Consistency is a *safety* property.

Definition 5 (Ledger Strong Liveness). A ledger \mathcal{L} satisfies strong liveness if for every valid transaction tx that is submitted by a client to all honest nodes is eventually included in the local ledger of every honest node. Formally, for any input tx proposed by a client to all honest nodes in \mathcal{H} by round r , there exists a round $r' \geq r$ such that for every $r'' \geq r'$ and for every honest node $i \in \mathcal{H}$, $\text{tx} \in \mathcal{L}_i^{r''}$.

Definition 6 (Ledger Security). A ledger is secure if it satisfies both consistency and strong liveness.

3.4 Adversarial Model

We assume that among the n nodes in the system, at most f may be faulty. Faulty nodes are controlled by a Byzantine adversary capable of arbitrary, potentially malicious behavior, including deviating from the protocol, colluding with other faulty nodes, or sending conflicting information to different honest nodes [13]. We model all parties—both honest nodes and the adversary—as Probabilistic Polynomial-Time (PPT) Turing machines with access to a source of uniform randomness. The PPT restriction bounds the computational power of both honest parties and the adversary to run in time polynomial in the size of their inputs.

The resilience threshold of deterministic BFT protocols—the maximum of Byzantine nodes f that can be tolerated—depends on the network model and the availability of authentication mechanisms. In the synchronous and partially synchronous model without Public Key Infrastructure (PKI), consensus is achievable for $f < n/3$ [14, 39], whereas in the asynchronous model, deterministic consensus is impossible for any $f > 0$ [10].

3.5 Client Model

We consider a sparse client V that is bootstrapping and connecting to the network for the first time. The client connects to a non-empty set \mathcal{P} of full nodes. We require that

the honest full nodes are secure, meaning their local ledgers are safe and live, as defined in Section 3.3.

We make two critical assumptions about the set \mathcal{P} :

- **Existential Honesty:** At least one node in \mathcal{P} is honest.
- **Non-Eclipsing:** This honest node is not eclipsed from the network after GST. Before GST, the honest node may be temporarily partitioned and unable to receive or send messages to other honest parties.

The rest of the nodes in \mathcal{P} can be controlled by an efficient byzantine adversary. In this model, the sparse client V acts as a verifier, while the full nodes act as provers [40, 41].

3.6 Sparse Node Model

A full node downloads, validates, and re-executes all transactions, maintaining a complete copy of the ledger and storing the entire state. In contrast, a *sparse node* downloads, validates and re-executes only a subset of transactions, maintaining a partial copy of the ledger, referred to as *sparse ledger*, and storing a corresponding subset of the global state, referred to as *sparse state* [3].

In this section, we formalize the notion of a sparse node and state some important security properties of a sparse node. Our work improves prior definitions introduced in [3], providing a more precise formulation of sparse ledgers and their corresponding sparse states.

3.6.1 Sparse Node Definition

We first model the state of a node. Let \mathcal{S}^r denote the state of a node at round r , represented as a function

$$\mathcal{S}^r : K \rightarrow V \quad (3.5)$$

mapping a set of keys K to a set of values V . Accordingly, a node stores a collection of key-value pairs (k, v) , where $k \in K$ is an identifier (e.g., the address of an account) and $v \in V$ represents the corresponding value (e.g., the state of an account).

A sparse node is defined via a *state predicate* \mathcal{X}_s , which, when applied to the keys of the global state \mathcal{S}^r at round r , selects a subset of keys. The resulting partial state is referred to as the *sparse state* $\hat{\mathcal{S}}^r$.

Definition 7 (State Predicate). A state predicate \mathcal{X}_s is a function

$$\mathcal{X}_s : K \rightarrow \{\top, \perp\} \quad (3.6)$$

that maps each key $k \in K$ to a boolean value. The predicate outputs \top if the sparse node is interested in the key, and \perp otherwise.

Using a state predicate, we can formally define the sparse state as the subset of the global state that a sparse node maintains.

Definition 8 (Sparse State). The sparse state $\hat{\mathcal{S}}^r \subseteq \mathcal{S}^r$ at round r is the subset of the global state \mathcal{S}^r , containing only the key-value pairs selected by the state predicate \mathcal{X}_s :

$$\hat{\mathcal{S}}^r = \{ (k, v) \in \mathcal{S}^r \mid \mathcal{X}_s(k) = \top \}. \quad (3.7)$$

We now formalize how transactions affect the current state. The state transition function δ , defined in Equation (3.2), takes as input the current state \mathcal{S}^r at round r and a transaction t , and outputs the updated state \mathcal{S}^{r+1} . Each transaction both reads from the current state and writes to the new state.

For a transaction tx applied to the state \mathcal{S}^r , we define two sets:

- The *read-set* of a transaction tx

$$\mathcal{R}(\mathcal{S}^r, \text{tx}) \subseteq \mathcal{S}^r \quad (3.8)$$

is the subset of the state entries in \mathcal{S}^r accessed via read operations.

- The *write-set* of a transaction tx

$$\mathcal{W}(\mathcal{S}^r, \text{tx}) \subseteq \mathcal{S}^r \quad (3.9)$$

is the subset of state entries in \mathcal{S}^r whose values will be modified by tx in the transition to the state \mathcal{S}^{r+1} .

To ease notation, in the remainder of this work we omit the explicit dependencies of the read and write sets on the state and transaction, and simply write \mathcal{R} and \mathcal{W} respectively whenever the context is clear. These definitions provide the basis for the *sparse state transition* $\hat{\delta}$.

Definition 9 (Sparse State Transition Function). At any round r , the sparse state transition function $\hat{\delta}$ is a deterministic function

$$\hat{\mathcal{S}}^{r+1} = \hat{\delta}(\hat{\mathcal{S}}^r \cup \mathcal{R}, \text{tx}) \quad (3.10)$$

where $\hat{\mathcal{S}}^r \cup \mathcal{R}$ denotes the union of the current sparse state $\hat{\mathcal{S}}^r$ and the read set \mathcal{R} of transaction tx . The function outputs the updated sparse state $\hat{\mathcal{S}}^{r+1}$ obtained by applying the transaction tx .

The sparse state transition function $\hat{\delta}$ can be seen as both a domain and range restriction of the global transition function δ . Concretely, $\hat{\delta}$ applies δ to its inputs in exactly the same way, but only returns those updated state elements k for which $\mathcal{X}_s(k) = \top$, i.e., those belonging to the sparse state.

In contrast to the standard state transition function δ , defined in Equation (3.2), which operates over the global state \mathcal{S} , the sparse state transition function $\hat{\delta}$ is restricted to the sparse state $\hat{\mathcal{S}}$ and the transaction's read set \mathcal{R} .

The explicit inclusion of the read set is essential because a transaction might read state entries $(k, v) \notin \hat{\mathcal{S}}$, and omitting them could lead to incorrect execution results. Therefore, in order to apply a transaction \mathbf{tx} correctly using the sparse state transition function $\hat{\delta}$, both \mathbf{tx} and its associated read set \mathcal{R} must be provided as input. This requirement arises from the fact that $\hat{\mathcal{S}}$ does not necessarily contain *all* relevant state entries and entails to direct implications (i) the read set incurs additional resource consumption, and (ii) the read set needs to be verified for correctness.

While the sparse state transition function $\hat{\delta}$ in Equation (3.10) describes how a sparse state $\hat{\mathcal{S}}$ is updated by applying a transaction, it does not specify *which* transactions from the global ledger \mathcal{L} are relevant for that specific state.

To address this, we introduce the *sparse ledger*, which represents an ordered subsequence of the global ledger that contains only the transaction of interest.

Definition 10 (Sparse Ledger). Let $\mathcal{L} = \langle \mathbf{tx}_1, \mathbf{tx}_2, \dots, \mathbf{tx}_m \rangle$ with $m \in \mathbb{N}$ denote a ledger, represented as the ordered sequence of transactions. A sparse ledger $\hat{\mathcal{L}}$ is any ordered subsequence of \mathcal{L} that preserves the relative order of the transactions in \mathcal{L} . Formally:

$$\hat{\mathcal{L}} = \langle \mathbf{tx}_{i_1}, \mathbf{tx}_{i_2}, \dots, \mathbf{tx}_{i_k} \rangle, \quad (3.11)$$

where $\{i_1, i_2, \dots, i_k\} \subseteq \{1, 2, \dots, m\}$ and $i_1 < i_2 < \dots < i_k$.

A sparse ledger provides a potentially partial view of the global ledger, which may be relevant to a particular substate of the global state.

When such a view is needed, one way to obtain it is by projecting the global ledger onto the subset of transactions that interact with a specified portion of the global state. This projection is formalized by a *transaction filtering operation* ϕ , which takes as input a state predicate \mathcal{X}_s and a ledger \mathcal{L} and produces a new ledger as output.

Definition 11 (Transaction Filter). Let \mathcal{X}_s be a state predicate. Let \mathcal{R} and \mathcal{W} denote the read and write set of a transaction, respectively. Let $\mathcal{L} = \langle \mathbf{tx}_1, \mathbf{tx}_2, \dots, \mathbf{tx}_m \rangle$ denote a ledger. The transaction filter ϕ is a function

$$\phi(\mathcal{X}_s, \mathcal{L}) = \langle \mathbf{tx} \mid \exists (k, v) \in \mathcal{R}(\mathbf{tx}) \cup \mathcal{W}(\mathbf{tx}) \wedge \mathcal{X}_s(k) = \top \rangle \quad (3.12)$$

which outputs a sequence containing only the transactions that read from or write to state entries satisfying the state predicate \mathcal{X}_s .

This sequence thus constitutes one possible instantiation of a sparse ledger, tailored to a specific substate defined by \mathcal{X}_s , while the general notion of a sparse ledger remains an arbitrary subsequence of the global ledger.

Remark 1 (Commutativity of the Transaction Filter). Note that the transaction filter ϕ is commutative. For any two state predicates \mathcal{X}_s^1 and \mathcal{X}_s^2 , and for any ledger \mathcal{L} , the following holds:

$$\phi(\mathcal{X}_s^1, \phi(\mathcal{X}_s^2, \mathcal{L})) = \phi(\mathcal{X}_s^2, \phi(\mathcal{X}_s^1, \mathcal{L})). \quad (3.13)$$

Both sides are equal to the set of transactions that touch at least one key relevant to the state predicate \mathcal{X}_s^1 and \mathcal{X}_s^2 .

Having established the foundational concepts of sparse ledgers, sparse state transition functions and their corresponding sparse state, we now turn to the security properties that ensure the correct and reliable operation of the system.

3.6.2 Sparse Node Security

To ensure correct and reliable operation when maintaining sparse views of the global state, it is essential to define the corresponding security properties. We say that a sparse node protocol Π is *secure* if it satisfies the following properties:

Definition 12 (Sparse Node Security). Consider a sparse node protocol $\Pi(\mathcal{P}, V(\mathcal{X}_s))$, characterized by a set of provers \mathcal{P} and a verifier V , parameterized by the state predicate \mathcal{X}_s . Let κ denote the protocol security parameter. The protocol is said to be secure in a partially synchronous network if, in every execution with at least one honest, non-eclipsed prover, except with probability $\text{negl}(\kappa)$, for every round r , the verifier V outputs a view $(\hat{\mathcal{L}}^r, \hat{\mathcal{S}}^r)$ that satisfies the following properties:

- **Consistency:** At every round r , the sparse ledger $\hat{\mathcal{L}}^r$ is a prefix of the filtered global ledger \mathcal{L} obtained via the transaction filter ϕ with respect to the state predicate \mathcal{X}_s . Formally:

$$\forall r : \hat{\mathcal{L}}^r \preceq \phi(\mathcal{X}_s, \mathcal{L}^r) \quad (3.14)$$

- **Liveness:** For every transaction tx in the global ledger \mathcal{L} such that $\mathcal{X}_s(k) = \top$, and for any round r in which tx first appears in \mathcal{L}^r of every honest node, there exists a round $r' \geq r$ such that, for all rounds $r'' \geq r'$, the transaction tx is included in the local sparse ledger $\hat{\mathcal{L}}^{r''}$. Formally:

$$\forall \text{tx} \in \mathcal{L}^r : \mathcal{X}_s(k) = \top \wedge \text{tx} \notin \mathcal{L}^{r-1} \implies \exists r' \geq r \forall r'' \geq r' : \text{tx} \in \hat{\mathcal{L}}^{r''} \quad (3.15)$$

A secure sparse ledger $\hat{\mathcal{L}}^r$ therefore eventually contains only those transactions from the global ledger \mathcal{L}^r that are relevant to the state predicate \mathcal{X}_s .

Theorem 1 (Consistency Across Sparse Ledgers). Let \mathcal{X}_s^1 and \mathcal{X}_s^2 denote two state predicates. Let $\hat{\mathcal{L}}_1$ and $\hat{\mathcal{L}}_2$ denote the corresponding sparse ledgers, derived from the same global ledger \mathcal{L} . If the sparse node protocols $\Pi_1(\mathcal{P}, V(\mathcal{X}_s^1))$ and $\Pi_2(\mathcal{P}, V(\mathcal{X}_s^2))$ are secure (i.e., consistent and live), then the projections of the two sparse ledgers onto their shared state are mutually consistent:

$$\phi(\mathcal{X}_s^2, \hat{\mathcal{L}}_1^r) \sim \phi(\mathcal{X}_s^1, \hat{\mathcal{L}}_2^r) \quad (3.16)$$

Proof. We prove Theorem 1 by leveraging security properties of the sparse node protocols and the determinism of the filtering operation ϕ .

Let \mathcal{L} denote the global ledger, and let $\hat{\mathcal{L}}_1^r$ and $\hat{\mathcal{L}}_2^r$ be the sparse ledgers maintained by the secure sparse protocols Π_1 and Π_2 , corresponding to state predicates \mathcal{X}_s^1 and \mathcal{X}_s^2 , respectively.

Since Π_1 and Π_2 are secure by assumption, each sparse ledger is consistent with the projection of the global ledger:

$$\begin{aligned}\hat{\mathcal{L}}_1^r &\preceq \phi(\mathcal{X}_s^1, \mathcal{L}^r), \\ \hat{\mathcal{L}}_2^r &\preceq \phi(\mathcal{X}_s^2, \mathcal{L}^r).\end{aligned}$$

Consider the projection of $\hat{\mathcal{L}}_1$ onto the transactions relevant to \mathcal{X}_s^2 . Since ϕ is deterministic and preserves the order of transactions, and because $\hat{\mathcal{L}}_1$ is a prefix of the filtered global ledger $\phi(\mathcal{X}_s^1, \mathcal{L})$, it follows that:

$$\phi(\mathcal{X}_s^2, \hat{\mathcal{L}}_1^r) \preceq \phi(\mathcal{X}_s^2, \phi(\mathcal{X}_s^1, \mathcal{L}^r)).$$

Similarly, consider the projection of $\hat{\mathcal{L}}_2$ onto the transactions relevant to \mathcal{X}_s^1 :

$$\phi(\mathcal{X}_s^1, \hat{\mathcal{L}}_2^r) \preceq \phi(\mathcal{X}_s^1, \phi(\mathcal{X}_s^2, \mathcal{L}^r)).$$

Since the transaction filter ϕ is commutative (see Remark 1), both projected sparse ledgers are prefixes of the same sequence. Consequently, one must be a prefix of the other, which was to be shown. \square

When the global ledger is safe and live, a secure sparse node protocol inherits both, consistency and liveness guarantees.

It is important to note that the security of a sparse node protocol ensures that, at every round r , the sparse state $\hat{\mathcal{S}}^r \subseteq \mathcal{S}^r$ is always a subset of the global state. Formally, for every key-value pair $(k, v) \in \hat{\mathcal{S}}^r$, there exists a corresponding pair $(k', v') \in \mathcal{S}^r$ such that if $k = k'$, then $v' = v$ holds. As a result, if the global state is valid, the validity of the sparse substate is likewise guaranteed.

When operating on a secure canonical chain, the validity of the sparse ledger follows directly from the validity of the global ledger. However, we now consider the case where an adversary controls more than $n/3$ validators, exceeding the $f < n/3$ resilience threshold of the underlying BFT protocol (see Section 3.4).

In this scenario, the sparse node protocol $\Pi(\mathcal{P}, V(\mathcal{X}_s))$ guarantees that its output—the sparse ledger $\hat{\mathcal{L}}$ and associated sparse state $\hat{\mathcal{S}}$ —maintains sparse validity. This property requires that every transaction in $\hat{\mathcal{L}}$ is valid with respect to the protocol's sparse state $\hat{\mathcal{S}}$.

Concretely, the verifier V checks that for every transaction tx , any state element (k, v) in its read set $\mathcal{R}(\text{tx})$ for which $\mathcal{X}_s(k) = \top$ is present and correct with respect to its local sparse state $\hat{\mathcal{S}}$. A block containing a transaction that violates this condition is rejected.

Consequently, under an adversarial majority where invalid transactions are injected into the global ledger, the protocol will accept transactions that represent valid state transitions with respect to its local sparse state, even if they are invalid transitions in the global state. Therefore, the validity property of the sparse ledger output by Π is weaker than the validity property of the ledger output by full nodes.

Finally, we note that a full view of the global ledger arises as a special case. If the state predicate \mathcal{X}_s always evaluates to \top , the sparse ledger coincides with the global ledger. In this case, the corresponding sparse state is identical to the global state.

Having formalized the sparse node model and its security properties, we now extend the discussion to event-based synchronization by introducing the event node model.

3.7 Event Node Model

While the sparse node model efficiently synchronizes a client with a filtered view of the ledger state, many practical blockchain applications require monitoring specific on-chain occurrences rather than tracking state transitions. Clients are often interested in a curated subset of the events emitted during transaction execution, such as token transfers to a specific address.

To formalize this paradigm of selective synchronization based on events, we introduce the *event node*. This node model is characterized by its maintenance of a stream of relevant events, filtered from the global event log according to a client-defined predicate.

Definition 13 (Event Predicate). An event predicate \mathcal{X}_{log} is a function

$$\mathcal{X}_{\text{log}} : \mathcal{E} \rightarrow \{\top, \perp\} \quad (3.17)$$

that maps each event $e \in \mathcal{E}$ to a boolean value. The predicate outputs \top if the event node is interested in the event, and \perp otherwise.

The complete, strictly ordered sequence of all events emitted by the execution of all transactions in the global ledger \mathcal{L} constitutes the global *event log*, denoted $\mathcal{L}_{\text{log}} = \langle e_1, e_2, \dots, e_n \rangle$. The order of events in this log is determined by the order of the corresponding transactions in the ledger and the order in which events are emitted within each transaction. A *sparse event log* is any subsequence of this log.

Definition 14 (Sparse Event Log). Let $\mathcal{L}_{\text{log}} = \langle e_1, e_2, \dots, e_m \rangle$ with $m \in \mathbb{N}$ denote an event log, represented as the ordered sequence of events. A sparse event log $\hat{\mathcal{L}}_{\text{log}}$ is any ordered subsequence of \mathcal{L}_{log} that preserves the relative order of the events in \mathcal{L}_{log} . Formally:

$$\hat{\mathcal{L}}_{\text{log}} = \langle e_{i_1}, e_{i_2}, \dots, e_{i_k} \rangle, \quad (3.18)$$

where $\{i_1, i_2, \dots, i_k\} \subseteq \{1, 2, \dots, m\}$ and $i_1 < i_2 < \dots < i_k$.

To construct this view, we define a filtering operation, ψ , which projects the global event log onto the subset of events satisfying the event predicate. Formally, the filtering operation ψ , takes as input an event predicate \mathcal{X}_{\log} and an event stream \mathcal{L}_{\log} and produces a new event log as output.

Definition 15 (Event Filter). Let \mathcal{X}_{\log} be an event predicate. Let \mathcal{L}_{\log} denote an event stream. The event filter ψ is a function

$$\psi(\mathcal{X}_{\log}, \mathcal{L}_{\log}) = \langle e \mid \mathcal{X}_{\log}(e) = \top \rangle \quad (3.19)$$

which outputs a sequence containing only the events satisfying the event predicate \mathcal{X}_{\log} .

If \mathcal{X}_{\log} is the constant \top predicate—that is, the predicate that accepts every event unconditionally—the output of ψ is trivially the entire global event log \mathcal{L}_{\log} .

3.7.1 Event Node Security

The security of an event node protocol is defined by its ability to provide a view of the event stream that is both consistent with and live relative to the global event log. Intuitively, consistency guarantees that the node’s log is a correct, unforged prefix of the filtered global log, while liveness ensures that all relevant events are incorporated into the node’s view within a bounded time.

Definition 16 (Event Node Security). Consider an event node protocol $\Pi(\mathcal{P}, V(\mathcal{X}_{\log}))$, characterized by a set of provers \mathcal{P} and a verifier V , parameterized by the event predicate \mathcal{X}_{\log} . Let κ denote the protocol security parameter. The protocol is said to be secure in a partially synchronous network if, in every execution with at least one honest, non-eclipsed prover, except with probability $\text{negl}(\kappa)$, for every round r , the verifier V outputs a view $\hat{\mathcal{L}}_{\log}$ that satisfies the following properties:

- **Consistency:** At every round r , the sparse event log $\hat{\mathcal{L}}_{\log}$ is a prefix of the filtered global event log \mathcal{L}_{\log} obtained via the event filter ψ with respect to the event predicate \mathcal{X}_{\log} . Formally:

$$\forall r : \hat{\mathcal{L}}_{\log}^r \preceq \psi(\mathcal{X}_{\log}, \mathcal{L}_{\log}^r) \quad (3.20)$$

- **Liveness:** For every event e in the global event log \mathcal{L}_{\log} such that $\mathcal{X}_{\log}(e) = \top$, and for any round r in which e first appears in \mathcal{L}_{\log}^r of every honest node, there exists a round $r' \geq r$ such that, for all rounds $r'' \geq r'$, the event e is included in the local sparse event log $\hat{\mathcal{L}}_{\log}^{r''}$. Formally:

$$\forall e \in \mathcal{L}_{\log}^r : \mathcal{X}_{\log}(e) = \top \wedge e \notin \mathcal{L}_{\log}^{r-1} \implies \exists r' \geq r \forall r'' \geq r' : e \in \hat{\mathcal{L}}_{\log}^{r''} \quad (3.21)$$

Therefore, a secure event node maintains a sparse event log $\hat{\mathcal{L}}_{\text{log}}^r$ containing only those events from $\mathcal{L}_{\text{log}}^r$ deemed relevant by \mathcal{X}_{log} , and eventually incorporates every such event.

Theorem 2 (Consistency Across Sparse Event Logs). Let $\mathcal{X}_{\text{log}}^1$ and $\mathcal{X}_{\text{log}}^2$ denote two event predicates. Let $\hat{\mathcal{L}}_{\text{log}}^1$ and $\hat{\mathcal{L}}_{\text{log}}^2$ denote the corresponding sparse event logs, derived from the same global event log \mathcal{L}_{log} . If the event node protocols $\Pi_1(\mathcal{P}, V(\mathcal{X}_{\text{log}}^1))$ and $\Pi_2(\mathcal{P}, V(\mathcal{X}_{\text{log}}^2))$ are secure (i.e., consistent and live), then the projections of the two sparse event logs onto their shared events are mutually consistent:

$$\psi(\mathcal{X}_{\text{log}}^2, \hat{\mathcal{L}}_{\text{log}}^1) \sim \psi(\mathcal{X}_{\text{log}}^1, \hat{\mathcal{L}}_{\text{log}}^2) \quad (3.22)$$

Proof. The proof follows directly from the proof of Theorem 1 in Section 3.6.2. We replace the transaction filter ϕ and state predicates \mathcal{X}_s with the event filter ψ and event predicates \mathcal{X}_{log} , respectively. Since ψ is deterministic, order-preserving, and commutative under composition, the same reasoning applies: the projections of two secure event node logs onto their shared event space are mutually consistent. \square



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Sparse Node Protocol

This chapter presents the two core protocols that enable lightweight clients to synchronize with a ledger or an event log in a secure and efficient manner. These lightweight clients, referred to as sparse node and event node, are designed to maintain only a subset of the global ledger, its corresponding state, or the global event log, tailored to the specific needs of individual clients and their applications. By leveraging the formally defined state and event predicates, alongside the transaction and event filtering operations (ϕ and ψ) introduced in Section 3.6, these clients decouple their resource consumption from the total size of the global ledger and event log, respectively.

We first outline the different modes of operation available to these lightweight clients, focusing on two representative cases: state-based and event-based. The fundamental challenge for any such client is to verify both the safety and liveness of all observed data relevant to its specific filtering criteria.

For state-based clients, we present *Sparseth*, a protocol for securely synchronizing the ledger and corresponding state. For contract accounts (i.e., accounts with code and storage), the protocol employs a dedicated counter mechanism to ensure that no relevant transactions have been omitted. For EOAs without code or storage, no such counter is required. Their correctness is fully attested through a commitment to the account state together with a Merkle proof. We provide explicit algorithms for both the verifier and the provers, detailing their respective roles in the protocol.

For event-based clients, we introduce *Eventeth*, a protocol that employs a cryptographic hash chain mechanism to enable the verification of both safety and liveness of the local event log. Here as well, we present algorithms for both the verifier and the provers, specifying how each party contributes to the protocol.

The protocols presented in this chapter offer several notable advantages. First, they impose zero overhead on consensus-layer validators. All commitments and verifications are handled within the execution layer, with commitments stored directly in the state of

the monitored smart contracts. This design requires no changes to the Ethereum protocol and stands in sharp contrast to validator-dependent approaches such as Sunfish [3], which require validators to explicitly include additional commitments in block headers. Second, both Sparseth and Eventeth are compatible with the asynchronous execution model [42]. Finally, by operating on predicate-defined subsets of data, the protocols drastically reduce computational, storage, and bandwidth requirements for clients.

Algorithm Notation Throughout the algorithms presented in this chapter, we assume a BFT ledger in which forks do not occur. The notation $m \dashrightarrow A$ denotes that a message m is sent to party A , while $m \dashleftarrow A$ indicates that message m is received from party A . Block headers are represented by the symbol B .

4.1 Modes of Operation

Lightweight clients, as introduced in this chapter, can operate in different modes. Each mode is characterized by the specific portion of the global ledger, global state, or global event log that they maintain. Importantly, the integrity of the maintained data is preserved unconditionally across all modes of operation. In this work, we focus on two representative modes, while acknowledging that additional modes have been explored in the literature [3] and may be explored as part of future work:

- **Event Node (Event-Based):** An event node is a specialized client that subscribes to and retrieves all events relevant to a predefined event predicate. These events are subsequently appended to a local, sparse event log. Crucially, and in contrast to a sparse node, an event node does not perform sparse execution, as events cannot be executed. This design renders the event node substantially more lightweight in terms of both computational and storage resources. Given the central role events play in modern blockchain ecosystems—supporting functionalities such as informing dApp frontends, enabling real-time contract auditing, or triggering off-chain automation workflows—a dedicated operational mode focused on events presents considerable practical utility.
- **Sparse Node (State-Based):** A sparse node is a client that synchronizes with the blockchain by selectively downloading, validating, and re-executing only those transactions relevant to a predefined state predicate. Consequently, it maintains a sparse state—a subset of the global state restricted to the keys defined by its state predicate. This operational mode is of particular interest as it effectively decouples the computational, storage, and bandwidth overhead of processing the entire blockchain from the workload associated with a specific state predicate, e.g., the set of smart contracts comprising the backend of a dApp. By retaining only the relevant fragment of the global state, sparse nodes facilitate scalable and efficient access to specific data, obviating the necessity for full-chain synchronization.

4.2 Eventeth

Events are a mechanism for smart contracts to emit structured data logs during execution, providing external applications with valuable insights into contract state and activity [29]. Eventeth, our event-based protocol, enables an event node to synchronize with the global event log by downloading only those events relevant to a predefined event predicate. These events are subsequently appended to a local, sparse event log, ensuring both consistency and liveness.

Operationally, an event predicate \mathcal{X}_{\log} (formally defined in Definition 13) is a function that specifies the subset of events a node is interested in tracking by evaluating to \top for any relevant event e . For certain applications, the criteria for an event predicate may be defined to align with specific state-based interests, but the predicate itself operates directly on the properties of the event. A canonical implementation often involves filtering events based on their emitting contract address.

For a practical example, consider the operational requirements of a front-end application for a Decentralized Exchange (DEX) such as Uniswap. An event node supporting such a system must capture all relevant on-chain activity, including **Mint** (liquidity addition), **Burn** (liquidity removal), or **Swap** (token exchange) events. These activities originate from a well-defined and finite set of smart contracts that implement the core functionality of the protocol, which we abstract as \mathcal{A}_{DEX} . In this context, the event predicate $\mathcal{X}_{\log}(e)$ is defined to select events emitted by a contract address k belonging to this set:

$$\mathcal{X}_{\log}(e) = \begin{cases} \top & \text{if } e \text{ was emitted by a contract address } k \text{ where } k \in \mathcal{A}_{\text{DEX}}, \\ \perp & \text{otherwise.} \end{cases} \quad (4.1)$$

This construction ensures that the event node retrieves precisely those events emitted by contracts relevant to the Uniswap protocol, enabling it to maintain a consistent and eventually complete application-level event log.

The utility of Eventeth critically depends on its ability to provide strong guarantees regarding both the safety and liveness of its local event log. To this end, the protocol must guarantee the security properties as formally defined in Definition 16 in Section 3.7.1. Safety ensures that any event in the local log is authentic and correctly ordered with respect to the global event log filtered by \mathcal{X}_{\log} . Liveness guarantees that all relevant events are eventually incorporated into the node's view within a bounded time.

To enforce these properties in a practical setting, Eventeth relies on a cryptographic hash chain mechanism. Specifically, the head of this hash chain is stored within the state of the monitored smart contract. Clients can independently verify the value of this commitment by reading the corresponding storage slot of the contract and validating it via a Merkle proof against the state root, which is committed in the block header. This design allows the client to independently confirm both the integrity (safety) and completeness (liveness) of the retrieved events without relying on external parties.

4.2.1 Hash Chain Mechanism

The theoretical constructs of an event predicate (Definition 13) and a sparse event log (Definition 14) require a practical cryptographic mechanism to ensure both the safety and liveness of the local event log. This mechanism is provided by a cryptographic hash chain, which serves as a commitment to the sequence of events emitted that satisfy the event predicate.

A hash chain, in its original form, is a sequence of values generated by the iterative application of a cryptographic hash function H to some piece of data d [43]. We extend this structure over a sequence of data items $\langle d_1, d_2, \dots, d_m \rangle$ with $m \in \mathbb{N}$ which is essential for capturing multiple events:

$$\begin{aligned} h_0 &= d_1 \\ h_i &= H(h_{i-1} \parallel d_i) \quad \text{for } i > 0 \end{aligned}$$

where \parallel denotes concatenation. This construction provides two critical properties for our use case. Firstly, the final value h_n serves as a succinct commitment to the entire, ordered sequence of data items $\langle d_1, d_2, \dots, d_m \rangle$. Secondly, any modification to any d_i or its position in the sequence will propagate through the chain, causing a mismatch in the final commitment h_m , thus reliably detecting omission, modification, and reordering. These properties directly leverage the binding property of cryptographic commitment schemes (as discussed in Section 2.2.3), ensuring that once a sequence of events is committed, it cannot be altered without detection.

In our architecture, each smart contract of interest maintains such a hash chain in its storage, where the individual data items d_i are the events it emits. The current head h_m of this chain thus constitutes a commitment to the contract's complete history of events. This capability is a prerequisite for the operation of the event node, enabling it to verify the integrity and completeness of any proposed event sequence against the on-chain commitment.

Hash-Chain of Events The recursive construction of the hash chain for a sequence of events $\langle e_1, e_2, \dots, e_m \rangle$ is defined by:

$$h_0 = c, \tag{4.2}$$

$$h_i = H(h_{i-1} \parallel e_i) \quad \text{for } i > 0 \tag{4.3}$$

Here, c is some fixed initialization constant (e.g., a zero-value), and H is a cryptographic hash function, as discussed in Section 2.2.2. By construction, each head h_i constitutes a succinct commitment to the entire prefix $\langle e_1, e_2, \dots, e_i \rangle$. In particular, the head h_b for a given block b cryptographically authenticates the complete sequence of all events emitted up to b , thereby providing a tamper-evident record resistant to insertion, omission, or modification.

Hash-Chain Verification To verify a proposed sequence of events $\mathcal{E} = \langle e_1, \dots, e_r \rangle$, a verifier (e.g., an event node) recomputes the hash chain iteratively. Starting from the known initialization constant c (Equation (4.2)), the verifier applies the recursive step (Equation (4.3)) for each event $e \in \mathcal{E}$. The sequence \mathcal{E} is accepted if and only if the final computed head h_r matches the canonical commitment.

4.2.2 Eventeth Protocol Mechanism

This section describes the operational mechanism of the Eventeth protocol for both the verifier and the prover. In this context, the event node operates as a verifier, while the provers are typically full nodes maintaining the entire blockchain.

Eventeth Verifier First, we describe the operational protocol run by an event node V for a given event predicate \mathcal{X}_{\log} . For clarity of presentation, we describe the protocol for a single key k which corresponds to a single smart contract. Thus, V maintains a single event log $\hat{\mathcal{L}}_{\log}$ and a single corresponding hash chain head h_l for k . The protocol trivially generalizes to multiple keys by instantiating separate event logs and independent hash chain heads for each contract. The complete procedure executed by V is specified in Algorithm 4.1.

Consider an event node V configured with an event predicate \mathcal{X}_{\log} and initialized with a known genesis state G and hash chain head h_{init} . Upon bootstrapping, V connects to a non-empty set of provers \mathcal{P} , one of which is assumed to be honest and not subject to an eclipse attack. The verifier then sends its event predicate \mathcal{X}_{\log} to all connected provers to specify the subset of events it is interested in.

Whenever a prover provides new data, the response contains:

- The block headers required to reconstruct the historical consensus parameters of the blockchain. In BFT-style consensus protocols, time is divided into epochs, with each epoch governed by a fixed validator set responsible for proposing and finalizing blocks. To enable efficient verification, the protocol periodically selects a subset of validators to form a sync committee, represented by their public keys. This committee collectively signs the block headers B , producing an aggregated signature π_B that attests to the validity of each block. Since validator stakes and committee memberships may change over time, the verifier must maintain an up-to-date view of the active committee. This is facilitated through handover messages, included in block headers, which specify the public keys of the next committee and allow verifiers to securely transition between committee configurations [38].
- An ancestry proof π_A that efficiently attests to the ancestry relationship between non-consecutive block headers. This proof is necessary because the client may not receive all intermediate blocks (e.g., it might obtain 5, 10, and 12, but not 6-9 or 11). The ancestry proof allows the verifier to confirm that the received blocks belong to the same fork of the chain without requiring all intermediate block

Algorithm 4.1: The Eventeth protocol algorithm run by the verifier V .

```

1 Function Verifier ( $\mathcal{X}_{\log}, \mathcal{P}, h_{\text{init}}$ ):
2    $\hat{\mathcal{L}}_{\log} \leftarrow \langle \cdot \rangle$ 
3    $h_l \leftarrow h_{\text{init}}$ 
4   foreach  $P \in \mathcal{P}$  do
5      $\mathcal{X}_{\log} \dashrightarrow P$ 
6     Run Verify ( $\mathcal{X}_{\log}, \hat{\mathcal{L}}_{\log}, h_l, P$ )           // Run concurrent for each prover
7   end
8
9 Function Verify ( $\mathcal{X}_{\log}, \hat{\mathcal{L}}_{\log}, h_l, P$ ):
10  while  $\top$  do
11     $\mathcal{D} \dashleftarrow P$ 
12     $(E, B, \pi_B, \pi_A, C, \pi_C) \leftarrow \mathcal{D}$ 
13    if IsValidBlockHeader ( $B, \pi_B, \pi_A$ ) =  $\perp$  then
14      | Disconnect ( $P$ )
15    end
16    if IsValidCommitment ( $C, \pi_C$ ) =  $\perp$  then
17      | Disconnect ( $P$ )
18    end
19    if IsValidSequence ( $\mathcal{X}_{\log}, E, C, h_l$ ) =  $\top$  then
20      |  $\hat{\mathcal{L}}_{\log} \leftarrow \hat{\mathcal{L}}_{\log} + E$ 
21      |  $h_l \leftarrow C$ 
22    else
23      | Disconnect ( $P$ )
24    end
25  end

```

headers [44]. Ancestry proofs are enabled by using Merkle Mountain Ranges, vector commitments, or skip lists –alternatively, if the chain does not support these proofs, all block headers can be sent, as in SPV clients.

- The sequence of filtered events E emitted during the execution of transactions in this block restricted to those for which $\mathcal{X}_{\log}(e) = \top$, together with the corresponding block header.
- A commitment C to the updated hash chain head, after transaction execution, together with a proof π_C attesting to the correctness of the commitment. By construction of the hash chain, this commitment also serves as a commitment to the entire event log $\hat{\mathcal{L}}_{\log}$ obtained so far.

For each prover $P \in \mathcal{P}$, the verifier starts an independent routine (Algorithm 4.1, Line 6) that executes the following procedure:

1. **Listening Phase:** The verifier V continuously listens for messages from a prover P (Algorithm 4.1, Line 11).
2. **Verification Phase:** Upon receiving a message $\mathcal{D} = (E, B, \pi_B, \pi_A, C, \pi_C)$ from P , V performs the following checks:
 - (i) V verifies π_B to ensure that B is a valid block header, and π_A to confirm that B belongs to the same fork of the chain (Algorithm 4.1, Line 13).
 - (ii) V verifies the commitment proof π_C in order to authenticate the value C provided by the prover (Algorithm 4.1, Line 16).
 - (iii) V verifies that every event $e \in E$ is relevant with respect to the event predicate \mathcal{X}_{\log} (Algorithm 4.2, Line 3), and that the event sequence E forms a valid log extension by reconstructing the hash chain for each $e \in E$ (Algorithm 4.2, Line 6). The complete event sequence verification procedure is detailed in Algorithm 4.2.
3. **Update Phase:** If all verifications succeed, V appends E to its local event log (Algorithm 4.1, Line 20), updates its local hash chain head (Algorithm 4.1, Line 21), and resumes waiting for the next message.
4. **Failure Handling:** If P provides invalid data (e.g., incorrect proofs or inconsistent events), V marks P as faulty and continues processing responses from the remaining provers (Algorithm 4.1, Lines 14, 17, 23). As long as at least one honest prover remains responsive, V can continue synchronizing its local event log.

The verifier V operates concurrently across all provers to tolerate Byzantine behavior. This design ensures liveness, as V does not block waiting for any single prover. Even if a majority of provers are malicious, a single honest prover is sufficient to keep the verifier updated. Note that the honest prover is only trusted for liveness—safety is guaranteed cryptographically. V only accepts data that passes all cryptographic verification, so any invalid data from a malicious prover is immediately rejected. If a malicious prover responds first with valid but stale data, it may be processed initially. However, the eventual response from an honest prover will contain the valid and current data, ensuring the verifier’s final state is correct. Thus, safety is derived from proof verification, while liveness depends on the existence of at least one honest prover.

Eventeth Prover We now present the protocol executed by an honest prover P within Eventeth. Unlike the verifier, the prover typically maintains global ledger \mathcal{L} and its associated state \mathcal{S} . In this context, the prover is responsible for filtering relevant events based on the event predicate \mathcal{X}_{\log} , constructing the corresponding cryptographic proofs, and publishing the resulting data to the verifier. The complete procedure executed by P is detailed in Algorithm 4.3.

During the bootstrap phase, the prover receives the event predicate \mathcal{X}_{\log} from the verifier V , which specifies the subset of events that V is interested in tracking.

Algorithm 4.2: Verification algorithm run by a verifier V to check the integrity of a sequence of events E against a commitment C . Here, h_l denotes the locally stored hash chain head so far.

```

1 Function IsValidSequence ( $\mathcal{X}_{\log}, E, C, h_l$ ):
2   foreach  $e \in E$  do
3     if  $\mathcal{X}_{\log}(e) = \perp$  then
4       return  $\perp$ 
5     end
6      $h_l \leftarrow H(h_l || e)$ 
7   end
8   if  $h_l = C$  then
9     return  $\top$ 
10  else
11    return  $\perp$ 
12  end

```

Algorithm 4.3: The Eventeth protocol algorithm run by provers P .

```

1 Function OnBootstrap():
2    $\mathcal{X}_{\log} \leftarrow V$ 
3
4 Function OnNewBlock ( $\mathcal{L}, \mathcal{S}, B, E, C, \mathcal{X}_{\log}$ ):
5    $\hat{E} \leftarrow \psi(\mathcal{X}_{\log}, E)$ 
6   if  $\hat{E} \neq \langle \cdot \rangle$  then
7      $\pi_A \leftarrow \text{ConstructProof}(\mathcal{L}, B)$ 
8      $\pi_B \leftarrow \text{ConstructProof}(\mathcal{L}, B)$ 
9      $\pi_C \leftarrow \text{ConstructProof}(\mathcal{L}, \mathcal{S}, C)$ 
10     $\mathcal{D} \leftarrow (\hat{E}, B, \pi_B, \pi_A, C, \pi_C)$ 
11     $\mathcal{D} \dashrightarrow V$ 
12  end

```

Upon arrival of a new block B , the prover performs the following steps:

1. **Event Filtering:** For each event $e \in E$ emitted during execution of the transactions in the current block, the prover computes the filtered sequence $\hat{E} = \psi(\mathcal{X}_{\log}, E)$ as introduced in Definition 15 (Line 5).
2. **Proof Construction:** After identifying the relevant events \hat{E} , the prover constructs the necessary proofs π_A , π_B , and π_C to convince the verifier of the correctness of the block header, filtered event sequence, and the commitment value (Lines 7-9).
3. **Data Publication:** Finally, the prover packages the filtered events, block header, commitment, and proofs into the message $\mathcal{D} = (\hat{E}, B, \pi_B, \pi_A, C, \pi_C)$ and sends it to the verifier (Lines 10-11). Note that data is published only if the filtered event sequence is non-empty, i.e., $\hat{E} \neq \langle \cdot \rangle$ (Line 6).

This process ensures that the verifier receives only the subset of events relevant to its registered predicates while still being able independently verify their correctness through cryptographic proofs.

4.3 Sparseth

While Eventeth focuses on maintaining a verifiable log of application-level events, *Sparseth*, our state-based protocol, is designed to efficiently track and verify the subset of the global ledger and corresponding state relevant to an individual client and her application. This approach allows clients to operate without processing or storing the entire global ledger and state, reducing resource requirements while maintaining strong security guarantees defined in Section 3.6.2—namely consistency with the global ledger, liveness, and sparse validity, even under adversarial majorities.

In practice, a state predicate defines the subset of the state keys that are relevant to a client. To illustrate this concept, we can again consider the operational requirements of a DEX like Uniswap. In this context, the state key k refers to the address of a smart contract. We define the state predicate \mathcal{X}_s for the Uniswap dApp as follows:

$$\mathcal{X}_s(k) = \begin{cases} \top & \text{if } k \in \mathcal{A}_{\text{DEX}} \\ \perp & \text{otherwise} \end{cases} \quad (4.4)$$

That is, \mathcal{X}_s outputs \top for any state key k (i.e., contract address) corresponding to a Uniswap core contract (such as the factory, pair, or specific liquidity pool contracts), and \perp otherwise. This predicate precisely captures the subset of the global state relevant to the protocol's operation. To reconstruct the application's evolving state, a sparse node must identify *all* transactions that access or modify the state of these contracts. This is achieved by applying the transaction filter ϕ (formally defined in Definition 11) which selects transactions that interact with state entries satisfying the \mathcal{X}_s . A transaction is

considered relevant if and only if it accesses or modifies the state of a contract k for which $\mathcal{X}_s(k) = \top$.

With this setup, a Uniswap-focused sparse node downloads, verifies, and re-executes only those transactions that interact with Uniswap's contracts, thereby building a sparse ledger and corresponding sparse state.

The utility of the Sparseth protocol critically depends on its ability to provide strong guarantees regarding the safety, liveness, and validity of its sparse ledger and corresponding sparse state. To this end, the protocol must guarantee the security properties as formally defined in Definition 12 of Section 3.6.2.

Safety ensures that the sparse ledger is a consistent prefix of the filtered global ledger. Consequently, the associated sparse state is always a correct subset of the global state, as it is derived from this ledger.

Liveness guarantees that all relevant transactions are eventually incorporated into the node's view, and the sparse state is updated within a bounded time.

These properties are enforced through distinct mechanisms tailored to different account types, as outlined in the introduction to this chapter. For accounts with associated code and storage (i.e., smart contracts), the protocol employs a dedicated counter mechanism to provide verifiable attestation that no relevant transactions have been omitted. For accounts without code or storage (i.e., EOAs), no such counter is required. Their state correctness is fully attested by verifying the account state against the global state root—a cryptographic commitment included in every block header—using a proof.

4.3.1 Counter Mechanism

Similar to the hash chain mechanism that enables verifiable liveness for event nodes, Sparseth requires a dedicated mechanism to ensure that no relevant transactions have been omitted when synchronizing the ledger and corresponding state. While transaction inclusion proofs—enabled by the state root commitment embedded in block headers—allow a verifier to confirm the correctness of individual transactions, they do not provide the client with any means to detect whether one or more relevant transactions have been censored or otherwise omitted. To address this limitation and guarantee liveness, we introduce an interaction counter mechanism for contract accounts.

Each smart contract of interest maintains a persistent, monotonically increasing integer counter. This counter is incremented upon every successful invocation of any public state-modifying function within the contract, and its value is stored as part of the contract's persistent state. Importantly, a single transaction may increment the counter by more than one—for instance, if another contract invokes multiple state-modifying functions of the contract with the sparse-node-compatible counter. The counter does *not* represent the number of transactions that touch the contract. This design is acceptable because the sparse node locally reconstructs the counter during selective transaction execution and only needs to match the final value stored in the contract's on-chain state.

Unlike the hash chain mechanism used in Eventeth (Section 4.2.1), which provides a cryptographic commitment to the sequence of events, the interaction counter in Sparseth is a simple integer value stored in the contract’s persistent state. The security guarantees of Sparseth rely on the combination of this counter together with transaction inclusion proofs. Specifically, the counter ensures completeness (no relevant transactions are omitted), while the inclusion proofs ensure correctness (each transaction is authentic and properly included in the block). Since the counter value forms part of the contract’s persistent storage, it is ultimately committed to the state root, which is itself included in the block header. Thus, a client can efficiently and verifiably read the counter from the blockchain state.

A sparse node, configured with a state predicate \mathcal{X}_s , locally re-executes transactions deemed relevant by the transaction filter ϕ . By comparing its locally reconstructed counter against the on-chain counter value, Sparseth can determine whether it has processed the complete set of relevant transactions. A match guarantees completeness, while a mismatch indicates that one or more transactions were missed.

For accounts without associated code and storage, no counter mechanism is required. In this case, the account balance, nonce, and other basic fields are directly committed to the state root. Thus, a cryptographic commitment to the account state, together with a Merkle proof, suffices to fully attest to the correctness of the account without requiring any additional mechanism.

4.3.2 Sparseth Protocol Mechanism

This section describes the operational mechanism of the Sparseth protocol for both the verifier and the prover. As before, the sparse node functions as the verifier, while the provers are typically full nodes maintaining the entire blockchain.

Sparseth Verifier First, we describe the operational protocol run by a sparse node V for a given state predicate \mathcal{X}_s . The complete procedure executed by V is specified in Algorithm 4.4.

Consider a sparse node V configured with a state predicate \mathcal{X}_s and initialized with a known sparse genesis state \hat{G} . Upon bootstrapping, V connects to a non-empty set of provers \mathcal{P} , one of which is assumed to be honest and not subject to an eclipse attack. The verifier then sends its state predicate \mathcal{X}_s to all connected provers to specify the subset of transactions it is interested in.

Whenever a prover provides new data, the response contains:

- The block headers required to reconstruct the historical consensus parameters of the blockchain (see Section 4.2.2).
- The corresponding ancestry proof π_A , which attests to the ancestor relationship between non-consecutive block headers (see Section 4.2.2).

Algorithm 4.4: The Sparseth protocol algorithm run by the verifier V .

```

1 Function Verifier ( $\mathcal{X}_s, \mathcal{P}$ ):
2    $\hat{S} \leftarrow \hat{G}$ 
3    $\hat{\mathcal{L}} \leftarrow \langle \cdot \rangle$ 
4   foreach  $P \in \mathcal{P}$  do
5      $\mathcal{X}_s \dashrightarrow P$ 
6     Run Verify ( $\mathcal{X}_s, \hat{\mathcal{L}}, \hat{S}_{cl}, P$ )           // Run concurrent for each prover
7   end
8
9 Function Verify ( $\mathcal{X}_s, \hat{\mathcal{L}}, \hat{S}, c_l, P$ ):
10  while  $\top$  do
11     $\mathcal{D} \dashrightarrow P$ 
12     $(B, \pi_B, \pi_A, T, \pi_I, R, C, \pi_C) \leftarrow \mathcal{D}$ 
13    if IsValidBlockHeader ( $B, \pi_B, \pi_A$ ) =  $\perp$  then
14      | Disconnect ( $P$ )
15    end
16    if IsValidCommitment ( $C, \pi_C$ ) =  $\perp$  then
17      | Disconnect ( $P$ )
18    end
19    foreach  $\text{tx} \in T$  do
20      | if IsTransactionIncluded ( $\text{tx}, \pi_I$ ) =  $\perp$  then
21        | | Disconnect ( $P$ )
22      | end
23      | if IsTransactionRelevant ( $\mathcal{X}_s, \hat{S}, \text{tx}, R[\text{tx}]$ ) =  $\perp$  then
24        | | Disconnect ( $P$ )
25      | end
26    end
27     $\hat{S}' \leftarrow \text{Snapshot}(\hat{S})$ 
28    foreach  $\text{tx} \in T$  do
29      |  $\hat{S} \leftarrow \hat{\delta}(\hat{S} \cup \mathcal{R}[\text{tx}], \text{tx})$ 
30    end
31    if IsValidState ( $\hat{S}, C$ ) =  $\top$  then
32      |  $\hat{\mathcal{L}} \leftarrow \hat{\mathcal{L}} + T$ 
33    else
34      |  $\hat{S} \leftarrow \text{Rollback}(\hat{S}')$ 
35      | Disconnect ( $P$ )
36    end
37  end

```

- The sequence of filtered transactions T within the block, restricted to those for which $\mathcal{X}_s(\text{tx}) = \top$. For each transaction tx the message also contains its corresponding read set, the block header in which it is included, and the associated inclusion proofs π_I .
- A commitment C to the updated interaction counter, after transaction execution, together with a proof π_C attesting to the correctness of the commitment. By construction, this commitment, combined with the inclusion proofs π_I also serves as a commitment to the current sparse ledger $\hat{\mathcal{L}}$.

The protocol is event-driven. For each prover $P \in \mathcal{P}$, the verifier starts an independent routine (Line 6) that executed the following procedure:

1. **Listening Phase:** The verifier V continuously listens for messages from all connected provers (Line 11).
2. **First Verification Phase:** Upon receiving a message $\mathcal{D} = (B, \pi_B, \pi_A, T, \pi_I, C, \pi_C)$ from P , V performs the following checks:
 - (i) V verifies π_B to ensure that B is a valid block header, and π_A to confirm that B belongs to the same fork of the chain (Line 13).
 - (ii) V verifies the commitment proof π_C to authenticate the value C (Line 16).
 - (iii) V verifies for each transaction $\text{tx} \in T$ that the inclusion proof π_I confirms the presence of tx in the claimed block (Line 20), and tx is relevant to the state predicate \mathcal{X}_s , i.e., it touches keys in the sparse state (Line 23).
3. **Execution Phase:** If all verifications succeed, V first creates a snapshot of its current sparse state $\hat{\mathcal{S}}$, allowing to roll back in case the prover is later found dishonest. Next, V executes each transaction tx by applying the sparse state transition function $\hat{\delta}$ (Definition 9) to the union of the transaction's read set $\mathcal{R}[\text{tx}]$ and the current sparse state $\hat{\mathcal{S}}$, updating its sparse state accordingly (Line 29).
4. **Second Verification Phase:** After executing all transactions $\text{tx} \in T$, V verifies whether P provided the complete set of transactions relevant to the state predicate \mathcal{X}_s (Line 31). To do so, V compares the state root derived from its updated sparse state and the interaction counter value against the values encoded in the provided commitment C .
 - If verification succeeds (\top), the prover supplied a complete and correct sequence of transactions. Consequently, V appends all transactions $\text{tx} \in T$ to its sparse ledger $\hat{\mathcal{L}}$ (Line 32) and resumes waiting for the next message.
 - If verification fails (\perp), the prover omitted one or more relevant transactions. The verifier then rolls back its state (Line 34) to the snapshot taken before, marks P as faulty and disconnects from it.

5. **Failure Handling:** If P sends invalid data at any stage (e.g., incorrect proofs or omits one or more transactions), V marks P as faulty and continues processing responses from the remaining provers (Lines 14, 17, 21, 24, 35). As long as at least one honest prover is available, V can continue synchronizing its local sparse ledger $\hat{\mathcal{L}}$ and computing the corresponding sparse state $\hat{\mathcal{S}}$.

Sparseth Prover We now present the protocol executed by an honest prover P within Sparseth. In this context, P operates as a full node with access to the global ledger \mathcal{L} and the corresponding global state \mathcal{S} . Leveraging this complete view, the prover filters transactions according to the state predicate \mathcal{X}_s , constructs the necessary cryptographic proofs, and publishes the resulting data to the verifier V . The complete procedure executed by P is detailed in Algorithm 4.5.

Algorithm 4.5: The Sparseth protocol algorithm run by provers P .

```

1 Function OnBootstrap():
2   |  $\mathcal{X}_s \leftarrow V$ 
3
4 Function OnNewBlock( $\mathcal{L}, \mathcal{S}, B, T, C, \mathcal{X}_s$ ):
5   |  $\pi_I \leftarrow \{\cdot\}$ 
6   |  $R \leftarrow \{\cdot\}$ 
7   |  $\hat{\mathcal{L}} \leftarrow \phi(\mathcal{X}_s, T)$ 
8   | foreach  $\text{tx} \in \hat{\mathcal{L}}$  do
9     |  $\pi_I \leftarrow \pi_I + \text{ConstructProof}(\text{tx})$ 
10    |  $R \leftarrow R + R[\text{tx}]$ 
11  | end
12  | if  $\hat{\mathcal{L}} \neq \langle \cdot \rangle$  then
13    |  $\pi_A \leftarrow \text{ConstructProof}(\mathcal{L}, B)$ 
14    |  $\pi_B \leftarrow \text{ConstructProof}(\mathcal{L}, B)$ 
15    |  $\pi_C \leftarrow \text{ConstructProof}(\mathcal{L}, \mathcal{S}, C)$ 
16    |  $\mathcal{D} \leftarrow (B, \pi_B, \pi_A, \hat{\mathcal{L}}, \pi_I, R, C, \pi_C)$ 
17    |  $\mathcal{D} \dashrightarrow V$ 
18  | end

```

During the bootstrap phase, the prover receives the state predicate \mathcal{X}_s from the verifier V , which specifies the subset of transactions that V is interested in executing.

Upon arrival of a new block B , the prover performs the following steps:

1. **Transactions Filtering:** For each transaction $\text{tx} \in T$ in the current block, the prover computes the filtered sequence $\hat{\mathcal{L}} = \phi(\mathcal{X}_s, T)$ as introduced in Definition 11 (Line 7).

2. **Proof Construction:** After identifying the relevant transactions $\hat{\mathcal{L}}$, the prover constructs the necessary proofs π_A , π_B , and π_I to convince the verifier of the correctness of the block header (Lines 13-14), transactions inclusion (Line 9), and the commitment value (Line 15).
3. **Read Set Construction:** For each transaction tx in $\hat{\mathcal{L}}$, the prover constructs the read set $R[\text{tx}]$ corresponding to the state keys accessed by tx (Line 10).
4. **Data Publication:** Finally, the prover packages the block header, the filtered transactions with their associated read sets, commitment, and proofs into the message $\mathcal{D} = (B, \pi_B, \pi_A, \hat{\mathcal{L}}, \pi_I, R, C, \pi_C)$ and sends it to the verifier (Lines 16-17).

This process ensures that the verifier receives only the subset of transactions relevant to its registered predicates while still being able independently verify their correctness through cryptographic proofs.

4.4 Discussion

In this section, we synthesize the implications of our work by examining the practical applications enabled by sparse client architectures, analyzing the fundamental trade-offs between our approach and existing solutions such as Sunfish, and outlining future directions as blockchain ecosystems continue to evolve. Having established the theoretical foundations of Sparseth and Eventeth in Section 3.6 and Section 3.7, respectively and presented their concrete protocol designs in Sections 4.3 and 4.2, we now situate their significance within the broader landscape of blockchain scalability and accessibility.

Sunfish [3] formalizes the notion of a sparse client and proposes validator-embedded commitments in block headers to enable sparse clients to reconstruct substates of interest. Our protocols share Sunfish’s high-level objective but adopt a distinct design that relocates commitments and verification to the execution layer.

Validator Overhead Sunfish requires validators to compute and include explicit commitments for supported substates in block headers. This model aligns well with blockchains like Sui, Solana, or Aptos, where validators perform heavy computation to enable lightweight clients. However, this introduces a validator cost that scales linearly with the number of supported predicates, potentially creating a bottleneck given the exponential number of possible substates. In contrast, Sparseth and Eventeth store commitments directly in contract storage and verify them against the canonical state root already embedded in block headers. This design fully eliminates additional consensus-layer overhead, ensuring that validators incur no extra cost regardless of the number or diversity of sparse substates that clients wish to follow.

Deployability Unlike Sunfish, which would require protocol modifications before deployment, Sparseth and Eventeth are immediately deployable on existing EVM-compatible

chains. Smart contracts that integrate the commitment logic instantly enable sparse client functionality, and existing developer tooling (e.g., RPC providers, indexers, and rollups) remains fully compatible. As a result, the path from prototype to real-world deployment is significantly shorter.

Costs Sunfish externalizes the majority of operational costs to validators and the consensus layer, maintaining extremely lightweight clients at the expense of systemic validator overhead. Our design adopts a different economic model by shifting a modest portion of costs to the execution layer through contract storage updates that maintain hash-chain heads or interaction counters. While this introduces marginal gas overhead borne by dApp users, this cost is economically viable on modern L2 rollups like Arbitrum or Optimism. We study further mitigation strategies as future work.

Access Lists for Sparse Clients Further ecosystem improvements promise to enhance the capabilities of sparse clients even further. For example, EIP-7928 [6] introduces the inclusion of a Block-Level Access List (BAL), a per-block record of all accounts and storage slots touched during transaction execution, together with their post-execution values, into the block header. By enforcing a complete access list, clients can perform parallel disk reads, execute transactions in parallel and even reconstruct post-state without transaction execution [6]. This enhancement directly complements the sparse client model, potentially enabling new sparse client designs that reconstruct the state of monitored accounts without storing full transaction data. If adopted, EIP-7928 would allow sparse clients to derive the state of monitored accounts directly from the BAL, effectively opening the door to a new class of sparse clients that do not store transactions at all while still being able to verifiably track the evolving state of their monitored contracts.

Notably, several related ecosystems already provide analogous primitives that expose per-transaction read sets. For example, Solana and Sui require transactions to declare the accounts or storage locations they will access in advance. This design facilitates static analysis of read sets and enables efficient sparse synchronization strategies. These developments suggest that as access-list mechanisms become more widespread, sparse client protocols such as *Sparseth* and *Eventeth* can be further simplified and accelerated.

4.4.1 Applications of Sparse Nodes

Running a node that verifies only a targeted subset of the blockchain unlocks use cases that are impractical for today's full nodes or light clients.

As identified in [3], sparse nodes are especially attractive to operators and users who need strong security guarantees while remaining cost-efficient. Examples include bridge operators, DAO token holders, participants in re-staking or remote-staking protocols, on-chain gaming platforms, and sequencers or watchers of rollups.

Beyond these client-side applications, sparse nodes can also strengthen the blockchain infrastructure itself. They can serve a portion of application-specific state to light clients, offloading data-serving duties from full nodes. They can maintain custom indexes for efficient on-chain data queries and help mitigate network congestion caused by heavily used contracts or regional traffic spikes.

Further use cases are enabled by our protocols. In our work, we introduced an even more lightweight type of node: the *event node*. Unlike sparse nodes that re-execute transactions, an event node maintains a log of specific on-chain events. This design is exceptionally resource-efficient, making it ideal for use cases that require reacting to proven on-chain activity without managing full state. For example, an event node could run inside a Trusted Execution Environment (TEE), monitor a contract for a specific on-chain event—such as a price-oracle update—and use the cryptographically verified result to trigger off-chain computation. This creates a powerful hybrid architecture that bridges on-chain trust with off-chain execution.

4.4.2 Advantages of Sparse Nodes

Sparse nodes present a middle ground between full nodes and light clients, combining verifiable security with practical resource efficiency. By monitoring only a subset of the blockchain, a sparse node isolates the workload of specific accounts from the workload of the entire blockchain.

Eventeth eliminates re-execution entirely for event-centric use cases by authenticating event sequences with a contract-maintained hash chain. Sparseth re-executes only transactions relevant to the monitored substate and verifies completeness via an interaction counter and inclusion proofs, avoiding the costs of full-chain execution. Despite their efficiency, Sparseth preserve strong security guarantees without relying on centralized NaaS providers. All accepted data is authenticated against commitments. This design restores user sovereignty over data access and improves privacy and censorship resistance compared to managed RPC endpoints. Sparseth is deployable today on EVM-compatible chains. Because commitments reside in contract storage, validators incur no extra protocol overhead, enabling incremental adoption on a per-application basis.

Sparseth and Eventeth are also *compatible with EVM-based rollups and asynchronous or pipelined execution architectures*. Separating verification and commitments from block production ensures that delayed execution does not compromise correctness or liveness, making the approach well-suited to high-throughput L2 ecosystems and emerging L1 chains.

Finally, sparse nodes complement the broader client ecosystem by serving authenticated, application-specific data to light clients and offloading data-serving duties from full nodes. They enable specialized indexes and reduce network load around heavily used contracts, improving overall responsiveness and resilience of the infrastructure.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Analysis

This chapter presents a rigorous analysis of the security properties of the Eventeth and Sparseth protocols, as introduced in Chapter 4. Our goal is to formally demonstrate that, under the assumption that the global ledger satisfies safety (Definition 4) and liveness (Definition 5), both Sparseth and Eventeth are secure as defined in Section 3.6.2 and Section 3.7.1, respectively, within our adversarial model (Section 3.4).

We begin by analyzing the Eventeth protocol and prove that it guarantees both consistency and liveness of the local event log, as defined in Definition 16. This is achieved by leveraging cryptographic hash chains, which ensure that events cannot be reordered, omitted, or tampered with by an adversary without detection.

Subsequently, we analyze the Sparseth protocol and show that it maintains a correct and live projection of the global state for its monitoring scope, as defined in Definition 12. Through its counter-based verification mechanism, Sparseth enables efficient detection of any omission of relevant transactions by a malicious prover, ensuring that the sparse state remains consistent with the underlying ledger.

We then present a theoretical resource analysis of both protocols, assessing their bandwidth, computation, and storage requirements to characterize their resource efficiency and establish asymptotic bounds on their operational costs.

Finally, we examine the implications of Data Availability (DA) attacks for sparse nodes. While sparse nodes can locally verify the correctness of their projected view of the ledger, they cannot guarantee that the entire block data has been published. This inherent limitation, shared with light clients, exposes them to adversarial strategies where a block producer selectively withholds transactions. However, in the case of Sparseth—unlike existing protocols such as Sunfish [3]—a successful DA attack requires a significantly stronger adversary, one who can precisely identify which transactions can be omitted without detection. Nevertheless, DA attacks remain an unavoidable

limitation for protocols that do not independently verify DA, highlighting the importance of complementary mechanisms.

5.1 Security Analysis of Eventeth

In this section, we formally analyze the security properties of the event node protocol as introduced in Section 4.2. Our focus is to establish that the protocol guarantees both consistency and liveness of the local event log maintained by the event node, under the adversarial model and cryptographic assumptions defined in Chapter 3. We show that, any adversarial attempt to forge, reorder, omit, or otherwise tamper with the sequence of relevant events can be effectively detected by the event node.

Theorem 3 (Eventeth Security). Eventeth is secure as per Definition 16.

Proof. Consistency: To prove consistency of Eventeth, we need to prove that, for every round r the sparse event log $\hat{\mathcal{L}}_{\log}^r$ output by the client is a prefix of the filtered global event log obtained via the event filter ψ with respect to the event predicate \mathcal{X}_{\log} , i.e.,

$$\forall r : \hat{\mathcal{L}}_{\log}^r \preceq \psi(\mathcal{X}_{\log}, \mathcal{L}_{\log}^r)$$

Towards this, consider the sparse event log $\hat{\mathcal{L}}_{\log}$ output by the client (Algorithm 4.1, Line 20). At each received block header B , the client tries to extend the $\hat{\mathcal{L}}_{\log}$ with the sequence E of events received by the provers (Algorithm 4.1, Line 12). Thereto, the client first validates that received block header is a valid ancestor of the stable tip of the chain (Algorithm 4.1, Line 13). Because the client only accepts a block header if it is valid and we consider secure blockchains operated by an honest supermajority of validator nodes, the commitment included in the block header is correct (Algorithm 4.1, Line 16). The client extends its local sparse event log only if the event sequence received correctly verifies against the hash chain head committed in C (Algorithm 4.1, Line 19). Thereto, the client locally extends the hash chain over the received sequence of events (Algorithm 4.2, Line 2–7) and verifies that the resulting hash chain head equals the hash chain head stored in the contract (Algorithm 4.2, Line 8), which is committed in C .

To complete the proof we now show that given a commitment C and a sequence of events $E = \langle e_1, e_2, \dots, e_n \rangle$, such that $H(E) = C$ (where $H(E)$ is shorthand notation for the hash chain extended over the events $e \in E$ according to Equation (4.3)) it is computationally infeasible to find a distinct sequence $E' = \langle e'_1, e'_2, \dots, e'_m \rangle$ with $E \neq E'$ such that $H(E') = C$. That is we show that if $H(E) = H(E') = C$, then it must hold that $E = E'$, except with probability $\text{negl}(\kappa)$.

Consider now an event predicate \mathcal{X}_{\log} . Because the client checks for every received event e that $\mathcal{X}_{\log}(e) = \top$ (Algorithm 4.2, Line 3), it is guaranteed that $\hat{\mathcal{L}}_{\log}$ only contains events that satisfy \mathcal{X}_{\log} , i.e., $\hat{\mathcal{L}}_{\log} \subseteq \psi(\mathcal{X}_{\log}, \mathcal{L}_{\log})$. It is left to show that the client receives all relevant events emitted in B , i.e., $\hat{\mathcal{L}}_{\log} \preceq \psi(\mathcal{X}_{\log}, \mathcal{L}_{\log})$. Because the client

locally reconstructs the hash chain (Algorithm 4.2, Line 6) and validates that its locally reconstructed head h_l equals the value committed in C (Algorithm 4.2, Line 8), and because of the collision resistance and preimage resistance properties of the hash function (Section 2.2.2), an adversary cannot find different preimages (i.e., a different sequence of events) whose hash yields C , unless with negligible probability in κ . It follows that $\hat{\mathcal{L}}_{\log}^r \preceq \psi(\mathcal{X}_{\log}, \mathcal{L}_{\log}^r)$ for all rounds r .

Liveness: To prove liveness of Eventeth, we need to prove that for every event e in the global event log \mathcal{L}_{\log} such that $\mathcal{X}_{\log}(e) = \top$, and for any round r in which e first appears in \mathcal{L}_{\log}^r of every honest node, there exists a round $r' \geq r$ such that, for all rounds $r'' \geq r'$ the event e is included in the local sparse event log $\hat{\mathcal{L}}_{\log}^{r''}$, i.e.,

$$\forall e \in \mathcal{L}_{\log}^r : \mathcal{X}_{\log}(e) = \top \wedge e \notin \mathcal{L}_{\log}^{r-1} \implies \exists r' \geq r \forall r'' \geq r' : e \in \hat{\mathcal{L}}_{\log}^{r''}$$

Because we assume that the client connects to at least one honest prover, the client is guaranteed to eventually receive updates for its sparse event log from that prover. Because we assume the prover is non-eclipsed, at any round $r \geq \text{GST}$ the event log of the honest prover extends the event log in the view of at least one honest validator. Because the prover runs Algorithm 4.3 for each new finalized block, it first filters its local event log (Algorithm 4.3, Line 5), generates the necessary proofs (Algorithm 4.3, Lines 7, 8, 9), and sends them to the verifier (Algorithm 4.3, Line 11). Because the communication is partially synchronous, for every round $r \geq \text{GST}$, the verifier is guaranteed to receive the update sent by the honest prover by $r + \Delta$, where Δ is the upper bound of the network on message transition time as discussed in Section 2.3.3. \square

5.2 Security Analysis of Sparseth

Having established the security properties of the event node, we now turn our attention to the state-based node. While both protocols aim to provide clients with strong guarantees about the validity and completeness of the data they maintain, the mechanisms and objects of interest differ: event nodes selectively download and verify events, whereas sparse nodes download, verify, and re-execute transactions.

In this section, we analyze the security guarantees provided by the sparse node protocol introduced in Section 4.3. We provide formal statements and proofs for the core properties: consistency and liveness. Our analysis demonstrates that, under the cryptographic assumptions as before, sparse nodes can reliably reconstruct the valid subset of the global state defined by the state predicate, ensuring both integrity and completeness of their local view.

Theorem 4 (Sparseth Security). Sparseth is secure as per Definition 12.

Proof. **Consistency:** To prove consistency of Sparseth, we need to prove that at every round r , the sparse ledger $\hat{\mathcal{L}}^r$ is a prefix of the filtered global ledger \mathcal{L} obtained via the

transaction filter ϕ with respect to the state predicate \mathcal{X}_s , i.e.,

$$\forall r : \hat{\mathcal{L}}^r \preceq \phi(\mathcal{X}_s, \mathcal{L}^r) \quad (5.1)$$

Towards this, consider the sparse ledger $\hat{\mathcal{L}}^r$ output by the verifier (Algorithm 4.4, Line 32). At each received block header B , the verifier tries to extend $\hat{\mathcal{L}}^r$ with the transactions received by the provers (Algorithm 4.4, Line 11). Thereto, the verifier first validates that received block header is a valid ancestor of the stable tip of the chain (Algorithm 4.4, Line 13). Because the client only accepts a block header if it is valid and we consider secure blockchains operated by an honest supermajority of validator nodes, the commitment included in the block header is correct (Algorithm 4.4, Line 16). The client only executes a received transaction tx if it is included in the block corresponding to block header B (Algorithm 4.4, Line 20) and $\text{tx} \in \phi(\mathcal{X}_s, \mathcal{L}^r)$ with respect to the verifier's state predicate \mathcal{X}_s (Algorithm 4.4, Line 23). Thus, $\hat{\mathcal{L}} \subseteq \phi(\mathcal{X}_s, \mathcal{L})$. It is left to show that the client receives all such transactions in B , i.e., $\hat{\mathcal{L}} \preceq \phi(\mathcal{X}_s, \mathcal{L})$. Thus, we need to show that the commitment C to the interaction counter together with the inclusion proofs of the respective transactions guarantees that every transaction tx for which $\text{tx} \in \phi(\mathcal{X}_s, \mathcal{L}^r)$ is re-executed.

The commitment C commits to a counter whose value corresponds to the total interactions with a state-modifying function of a contract account that is relevant with respect to \mathcal{X}_s . The inclusion proof π_I commits to the transaction tx included in B . Together, they ensure $\hat{\mathcal{L}} \preceq \phi(\mathcal{X}_s, \mathcal{L})$. Because of the collision resistance and preimage resistance properties of the hash function (Section 2.2.2), an adversary cannot find different preimages (i.e., a different sequence of transactions) whose hash yields C and π_I unless with negligible probability in κ . Because we consider secure blockchains, we know that these commitments are correct. Because each relevant transaction is executed using the sparse state transition function (Algorithm 4.4, Line 29) defined in Definition 9 which applies the global state transition function δ to its inputs in exactly the same way, but only returns those updated state elements k that satisfy the state predicate ($\mathcal{X}_s(k) = \top$), it is guaranteed that each k is updated exactly as in the global state. Because the correct amount of counter additions is checked (Algorithm 4.4, Line 31), the appended sequence of transactions is guaranteed to be complete, and therefore $\hat{\mathcal{L}}^r \preceq \phi(\mathcal{X}_s, \mathcal{L}^r)$.

Liveness: To prove liveness of Sparseth, we need to prove that for every transaction tx in the global ledger \mathcal{L} such that $\mathcal{X}_s(k) = \top$, and for any round r in which tx first appears in \mathcal{L}^r of every honest node, there exists a round $r' \geq r$ such that, for all rounds $r'' \geq r'$, the transaction tx is included in the local sparse ledger $\hat{\mathcal{L}}^{r''}$, i.e.,

$$\forall \text{tx} \in \mathcal{L}^r : \mathcal{X}_s(k) = \top \wedge \text{tx} \notin \mathcal{L}^{r-1} \implies \exists r' \geq r \forall r'' \geq r' : \text{tx} \in \hat{\mathcal{L}}^{r''}$$

Because we assume that the client connects to at least one honest prover, the client is guaranteed to eventually receive updates for its sparse ledger from that prover. Because we assume the prover is non-eclipsed, at any round $r \geq \text{GST}$ the ledger of the honest prover extends the ledger in the view of at least one honest validator. Because the

prover runs Algorithm 4.5 for each new finalized block, it first filters its local ledger (Algorithm 4.5, Line 7), generates the transaction read sets (Algorithm 4.5, Lines 8) as well as the necessary proofs (Algorithm 4.5, Lines 9, 13, 14, 15), and sends them to the verifier (Algorithm 4.5, Line 17). Because the communication is partially synchronous, for every round $r \geq \text{GST}$, the verifier is guaranteed to receive the update sent by the honest prover by $r + \Delta$, where Δ is the upper bound of the network on message transition time as discussed in Section 2.3.3. \square

5.3 Analysis of Sparse Node Resource Consumption

Having defined the operational protocols for both sparse and event nodes (Section 4.3 and Section 4.2, respectively), we now analyze their efficiency and resource implications. This section quantifies the advantages of our proposed lightweight client architectures, Sparseth and Eventeth, by discussing the resources consumed in terms of bandwidth, computation, and storage.

5.3.1 Event Node Resource Consumption

An event node, running the Eventeth protocol, is designed to efficiently monitor and process specific on-chain events. To achieve this, the event node synchronizes with the canonical blockchain by downloading all necessary block headers. This includes headers for consensus purposes (e.g., those containing sync committee handover messages) as well as headers for blocks that contain events matching its predicate. Subsequently, it selectively retrieves only those events that satisfy a predefined event predicate, filtering the global event stream according to application-specific criteria.

This design specifically serves use cases that depend on reliable event log data but for which the overhead of processing entire transactions is impractical.

- **Bandwidth:** The bandwidth of an event node running the Eventeth verifier protocol is driven by three components:
 - Downloading a subset of block headers to track the canonical chain, e.g., those containing sync committee handover messages ($\mathcal{O}(\lambda|\mathcal{L}|)$).
 - Downloading the headers of blocks that contain relevant events ($\mathcal{O}(\rho_e|\mathcal{L}|)$).
 - Retrieving the event logs from those relevant blocks ($\mathcal{O}(|\hat{\mathcal{L}}_{\log}|)$).

The total bandwidth is therefore:

$$\mathcal{O}(\lambda|\mathcal{L}| + \rho_e|\mathcal{L}| + |\hat{\mathcal{L}}_{\log}|),$$

where λ is the fraction of blocks whose headers are downloaded for consensus, ρ_e is the fraction of blocks that contain events matching the predicate \mathcal{X}_{\log} , and $|\hat{\mathcal{L}}_{\log}|$ is the total size of the filtered event data.

This bandwidth profile is higher than a pure light client's $\mathcal{O}(\lambda|\mathcal{L}|)$ but substantially less than a full node's $\mathcal{O}(|\mathcal{L}|)$ requirement for $\lambda \ll 1$ and $\rho \ll 1$.

- **Computation:** The computational requirements for an event node are primarily determined by two operations. First, the node must reconstruct the hash chain for each contract it tracks, which involves performing cryptographic hash computations over the relevant events extracted from the blockchain. Second, the node must verify the integrity of these reconstructed hash chain heads by comparing them to the corresponding on-chain commitments, typically using succinct cryptographic proofs. The total computational complexity for these operations is:

$$\mathcal{O}(|\hat{\mathcal{L}}_{\log}|),$$

as each event in the filtered event log must be individually processed. This computational burden is significantly lower than that of a full node, which is required to execute all transactions in the global ledger—an operation with complexity $\mathcal{O}(|\mathcal{L}| + |\mathcal{S}|)$. However, it is higher than the cost for many light client designs. While some light clients achieve a constant verification cost $\mathcal{O}(1)$ [38], others (like traditional SPV clients [22]) may incur linear costs $\mathcal{O}(|\mathcal{L}|)$ for verifying transaction inclusion in the longest chain.

- **Storage:** The storage requirements for an event node consist of three main components: (i) the block headers of the canonical chain, requiring $\mathcal{O}(\lambda|\mathcal{L}|)$ storage, (ii) the local sparse event log, which contains only the filtered events relevant to the node's predicate, requiring $\mathcal{O}(|\hat{\mathcal{L}}_{\log}|)$ storage, and (iii) the set of hash chain heads, with one head stored for each tracked contract, which is accounted for in $\mathcal{O}(|\hat{\mathcal{L}}_{\log}|)$. The total storage requirement is therefore:

$$\mathcal{O}(\lambda|\mathcal{L}| + |\hat{\mathcal{L}}_{\log}|),$$

This consolidated storage for block headers, filtered event data, and hash chain heads is greater than that of a light client, but remains significantly lower than the full node requirement of $\mathcal{O}(|\mathcal{L}| + |\mathcal{S}|)$, especially when the set of tracked accounts and the event log are small relative to the global ledger and state. This enables event nodes to efficiently monitor and verify specific on-chain activity with minimal storage overhead.

5.3.2 Sparse Node Resource Consumption

A sparse node, implementing the Sparseth protocol, maintains a partial view of the global ledger and a corresponding sparse state by selectively downloading and re-executing only those transactions that are relevant to a predefined state predicate \mathcal{X}_s . To quantify the resource consumption of a sparse node relative to the global ledger and state, we introduce two scalar parameters:

- Let β (with $0 \leq \beta \leq 1$) be the fraction of the key-value entries of the global state that are relevant to the sparse node, as defined by the state predicate.
- Let α (with $0 \leq \alpha \leq 1$) denote the fraction of the global ledger (i.e., transactions) that is relevant to the sparse node. In general, α and β do not need to coincide, as the fraction of relevant transactions does not have to match the fraction of relevant state entries. However, there is a relationship between them: when $\beta = 1$, then $\alpha = 1$, and when $\beta = 0$, then $\alpha = 0$.

Thus, the size of the sparse ledger and corresponding sparse state can be expressed as:

$$|\hat{\mathcal{L}}| = \alpha|\mathcal{L}| \quad \text{and} \quad |\hat{\mathcal{S}}| = \beta|\mathcal{S}|$$

These parameters define a spectrum of possible behaviors for a sparse node. The extreme cases at the boundaries of α and β are particularly instructive, as they reflect the behaviors of well-known client types:

- **Full replication** ($\alpha = 1, \beta = 1$): In the scenario where the sparse node is configured to be interested in all state entries, as specified by the trivial state predicate $\forall k \in \mathcal{S} : \mathcal{X}_s(k) = \top$, every transaction within the ledger is deemed relevant. Consequently, the node must download and execute the entirety of the ledger to maintain a complete view of the global state. Under these conditions, the operational behavior of the sparse node is indistinguishable from that of a traditional full node, as it replicates both the global ledger and the corresponding global state in their entirety.
- **Header-only verification** ($\alpha = 0, \beta = 1$): In contrast, when the sparse node is configured with the trivial state predicate $\forall k \in \mathcal{S} : \mathcal{X}_s(k) = \perp$, it expresses no interest in any state entries. As a result, no transactions are deemed relevant to its operation. Under these conditions, the node restricts its synchronization activity to the download of block headers alone, thereby tracking the canonical chain without processing any transaction or state data. This operational mode is functionally equivalent to that of a conventional light client, as the node neither maintains a local ledger nor reconstructs any portion of the global state.

While the examination of these limiting cases is instructive, the protocol is principally optimized for scenarios in which $\beta \ll 1$. In this regime, sparse nodes achieve significant efficiency improvements by synchronizing only a small subset of the global ledger and state. As β approaches 1, the efficiency gains diminish, with resource requirements converging to those of a full node while incurring additional protocol complexity. Under such circumstances, the operation of a full node is both simpler and more cost-effective. Consequently, the sparse node architecture is most beneficial when the relevant portion of the state is substantially smaller than the global state.

We now turn to a detailed discussion of the resource requirements for this client type.

- **Bandwidth:** The bandwidth requirements of a sparse node running the Sparseth verifier protocol is driven by three components:
 - Downloading a subset of block headers to track the canonical chain, e.g., those containing sync committee handover messages ($\mathcal{O}(\lambda|\mathcal{L}|)$).
 - Downloading the headers of blocks that contain relevant transactions ($\mathcal{O}(\rho|\mathcal{L}|)$).
 - Retrieving the transactions from those relevant blocks ($\mathcal{O}(\alpha|\mathcal{L}|)$).

Formally, the asymptotic bandwidth consumption can be expressed as:

$$\mathcal{O}(\lambda|\mathcal{L}| + \rho_s|\mathcal{L}| + \alpha|\mathcal{L}|),$$

where λ is the fraction of blocks whose headers are downloaded for consensus, ρ_s is the fraction of blocks that contain transactions matching the predicate \mathcal{X}_s , and $\alpha|\mathcal{L}|$ captures the additional bandwidth needed to download the filtered transactions and their associated read-sets.

For $\alpha \ll 1$, this total bandwidth represents a substantial reduction when compared to a full node. However, the requirement still exceeds that of a light client whose bandwidth is limited to $\mathcal{O}(\lambda|\mathcal{L}|)$ due to the additional need for transaction and read-set data.

- **Computation:** The computational complexity incurred by a sparse node arises from the execution of all transactions identified as relevant by its state predicate, as well as from the access and manipulation of the corresponding subset of the sparse state during transaction processing. Specifically, this entails the application of the sparse state transition function $\hat{\delta}$ to each relevant transaction, in addition to the verification of on-chain state commitments and interaction counters, which collectively uphold the protocol's safety and liveness guarantees. Crucially, we assume that the computational cost of processing a single transaction is bounded, i.e., a transaction cannot trigger infinite computation. The aggregate computational complexity can be characterized as:

$$\mathcal{O}(\alpha|\mathcal{L}| + \beta|\mathcal{S}|),$$

where the first term reflects the cost associated with processing $\alpha|\mathcal{L}|$ relevant transactions, and the second term accounts for operations over $\beta|\mathcal{S}|$ state entries. While this computational burden exceeds that of light or event nodes, it remains substantially lower than the full node requirement of $\mathcal{O}(|\mathcal{L}| + |\mathcal{S}|)$ in the regime where $\alpha \ll 1$ and $\beta \ll 1$.

- **Storage:** A sparse node stores two primary data components: its local sparse ledger and the corresponding sparse state. The sparse ledger, containing $\alpha|\mathcal{L}|$ relevant transactions, requires storage proportional to $\mathcal{O}(\alpha|\mathcal{L}|)$. The sparse state, comprising $\beta|\mathcal{S}|$ relevant key-value entries, requires storage proportional to $\mathcal{O}(\beta|\mathcal{S}|)$. The total storage requirement for a sparse node is therefore:

$$\mathcal{O}(\lambda|\mathcal{L}| + \alpha|\mathcal{L}| + \beta|\mathcal{S}|).$$

Client Type	Bandwidth	Computation	Storage
Full Node	$\mathcal{O}(\mathcal{L})$	$\mathcal{O}(\mathcal{L} + \mathcal{S})$	$\mathcal{O}(\mathcal{L} + \mathcal{S})$
Sparse Node	$\mathcal{O}(\lambda \mathcal{L} + \rho_s \mathcal{L} + \alpha \mathcal{L})$	$\mathcal{O}(\alpha \mathcal{L} + \beta \mathcal{S})$	$\mathcal{O}(\lambda \mathcal{L} + \alpha \mathcal{L} + \beta \mathcal{S})$
Event Node	$\mathcal{O}(\lambda \mathcal{L} + \rho_e \mathcal{L} + \hat{\mathcal{L}}_{\log})$	$\mathcal{O}(\hat{\mathcal{L}}_{\log})$	$\mathcal{O}(\lambda \mathcal{L} + \hat{\mathcal{L}}_{\log})$
Light Client	$\mathcal{O}(\lambda \mathcal{L})$	$\mathcal{O}(\mathcal{L})$	$\mathcal{O}(\lambda \mathcal{L})$

Table 5.1: Asymptotic resource consumption for different client types.

When both α and β are much less than 1, the resulting storage requirements for a sparse node are substantially reduced relative to those of a full node. This reduction reflects the protocol’s ability to limit local data retention strictly to the subset of ledger entries and state elements deemed relevant, thereby achieving significant efficiency gains in storage utilization.

Beyond their primary function as clients, the architecture of sparse nodes also enables them to serve data within the network infrastructure. By distributing data-serving responsibilities, they reduce load on full nodes. To fulfill this role, a sparse node must act as a prover, persistently storing the cryptographic proofs it receives. Crucially, the size of these proofs is bounded—for instance, a Merkle proof grows only logarithmically with the size of the authenticated state. This keeps storage overhead manageable as the network scales and enables sparse nodes to serve client queries efficiently without resource-heavy re-fetching or recomputation.

This section has quantified the resource consumption of event nodes running the Eventeth protocol and sparse nodes running the Sparseth protocol respectively, detailing the bandwidth, computational, and storage requirements. Our analysis demonstrates that by focusing on predicate-defined subsets of data, both protocols can achieve a significant reduction in resource overhead compared to the traditional full node model. A comparative overview, including traditional full nodes and light clients, is presented in Table 5.1. Note that this analysis focuses on the costs of synchronization and verification. The optional role of serving proofs to other clients would introduce additional, bounded storage overhead proportional to the sparse ledger, not reflected in Table 5.1.

5.4 Data Availability Attacks

While full nodes download and re-execute all transactions in every block, light clients—for scalability reasons—download and verify only block headers and rely on cryptographic proofs to verify the inclusion of specific transactions of interest (see Section 2.6.5). While this approach drastically reduces resource requirements, it introduces the Data Availability (DA) problem for light clients.

The DA problem arises when a malicious block producer publishes the block header, but withholds some portion of the block’s data—for instance invalid transactions. For full nodes, this is not a problem: When they fail to retrieve the complete transaction set, they can simply reject the block and refuse to propagate it further. However, without access to the missing data, they cannot generate fraud proofs, demonstrating the invalidity of the block. In contrast, light nodes cannot detect the data unavailability and may accept the block, even if critical parts of the data were never published [45, 46].

A further complication is that even if honest full nodes detect data unavailability, alerting light nodes is unreliable. An adversarial block producer may release the missing data after becoming aware of the alert, or network delays may cause clients to receive the withheld data before the warning is propagated [45].

Consequently, light clients face inherent risks in the absence of additional mechanisms introduced in the literature, such as [45, 46, 37].

5.4.1 Implications for Sparse Nodes

Sparse nodes, unlike full nodes, do not download and execute all transactions of every block. Instead, they selectively download and re-execute only those transactions that match a predefined state predicate (see Section 3.6). This design makes them more efficient, but it also means that sparse nodes cannot verify whether the *entire* block has been published.

As a consequence, sparse nodes inherit the DA problem: A malicious block producer can withhold parts of the block data that are irrelevant to the sparse node’s predicate but still necessary for the integrity of the global state. From the perspective of the sparse node, the block may appear complete and internally consistent, since all predicate-matching transactions are available and verifiable, even though other parts of the block remain unpublished. Thus, while sparse nodes guarantee correctness of their local projection, they cannot ensure completeness of the global block data and must rely on honest full nodes to enforce DA.

However, the practical feasibility of a targeted attack differs between existing systems such as Sunfish [3] and our proposed Sparseth protocol. In Sunfish, block producers are explicitly aware of the monitored state predicates, since they must include commitments in the block header. This gives an adversary precise knowledge of which transactions are required for sparse clients, making selective withholding straightforward.

In contrast, Sparseth introduces a higher barrier for an adversary. Our protocol does not expose any predicate commitments in block headers. Instead, interaction counters and hash chain heads are stored within the smart contract’s state itself. Block producers have no direct insight into the predicates tracked by individual sparse nodes. To perform a targeted attack, an adversary would need to (i) scan and analyze the entire execution layer to identify smart contracts relevant to potential victims, (ii) reverse-engineer or guess the state predicates that sparse nodes use, and (iii) correctly identify and withhold

only those transactions that are irrelevant to the victim's predicate. The lack of on-chain commitments in Sparseth makes targeted DA attacks more difficult to execute at scale compared to protocols with explicit predicate commitments.

Several mitigation techniques have been proposed in the literature, such as data availability sampling [47]. While these approaches also strengthen sparse nodes against DA attacks, the integration of such mechanisms is beyond the scope of this work.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Implementation

This chapter presents the practical implementation of the sparse node protocol, translating the theoretical concepts introduced in Section 3.6 and Chapter 4 into a practical system.

It begins by detailing the necessary adaptations to Solidity smart contracts for sparse monitoring, namely the interaction counter (introduced in Section 4.3) and the hash chain mechanism (introduced in Section 4.2). Following the contract-level modifications, the chapter introduces the architecture of the Proof of Concept (PoC) sparse node client developed in Go. The client leverages the go-ethereum library and the Ethereum JSON-RPC API to realize the protocol. A central focus is on the main technical challenges encountered and the practical solutions and trade-offs adopted to address the issues.

The full PoC implementation is publicly available on GitHub at <https://github.com/pslowak/sparseth> and released under the MIT license, ensuring transparency, reproducibility, and extensibility for future research and development.

By bridging abstract protocol design and concrete deployment, this chapter provides a comprehensive overview of the engineering required to realize sparse blockchain monitoring in practice.

6.1 Smart Contract Adaption for Sparse Monitoring

As established in the protocol design (Chapter 4), enabling sparse monitoring requires the implementation of two core verification primitives: a counter mechanism (Section 4.3) and hash chain mechanism (Section 4.2).

This section details the practical implementation of these primitives. The EVM executes bytecode. While it is possible to write smart contracts directly in bytecode, it is highly uncommon, since bytecode is hard to read and understand. Instead, most programmers rely on high-level programming languages to write programs and compile it into bytecode.

Our implementation focuses on EVM-compatible smart contracts written in Solidity. Solidity is a statically-typed, and Turing-complete programming language designed for development of EVM-based applications. While alternative languages such as LLL, Serpent, or Vyper exist, Solidity remains the most prominent and widely adopted language for Ethereum smart contract development [48, 49, 50].

Focusing on Solidity smart contracts provides several practical benefits, as it allows us to: (i) leverage standard interfaces (e.g., ERC-20 tokens), (ii) exploit indexed events, and (iii) target the largest and most active smart contract ecosystem. Consequently, Solidity provides an ideal platform for demonstrating the feasibility of sparse node monitoring in a realistic setting. All implementations presented in this work are based on Solidity version 0.8.30.

6.1.1 Counter Mechanism Implementation

This section describes the practical integration of the counter mechanism into Solidity smart contracts. The primary goal is to ensure that every state-modifying transaction is recorded, providing a foundation for verifiable completeness. The implementation must adhere to two key aspects of Solidity’s function design: function visibility and state mutability.

Function Visibility The counter must be incremented for every externally accessible function that alters the contract’s state. In Solidity, function visibility specifiers define how functions can be called. Solidity supports two kinds of function calls: external calls, which generate an actual EVM message call, and internal calls, which are resolved within the same contract and do not incur the overhead of an external call. Additionally, internal calls can be restricted from derived contracts. Combining these options, Solidity defines four visibility levels for functions [51]:

- **external:** Functions marked as `external` can be invoked from other contracts or via transactions. They cannot be called internally without using the `this` keyword, as doing so would require an explicit message call.
- **public:** Functions marked as `public` are the most permissive. They can be called both internally and externally, making them accessible from any context.
- **internal:** Functions marked as `internal` can only be called from within the current contract or from contracts that inherit from it.
- **private:** Functions marked as `private` are the most restrictive. They are only accessible within the defining contract and are not visible to derived contracts.

Since the counter must be incremented at least once per transaction touching the contract, a common pattern is to update it at the start of every `public` or `external` function that modifies the state. This guarantees that every state change is reflected in the counter, while avoiding unnecessary updates during purely internal computations.

```

1 contract Storage {
2
3     uint256 private _counter;
4
5     uint256 private _value;
6
7     function store(uint256 val) external {
8         _counter++;
9         _value = val;
10    }
11
12    function retrieve() public view returns (uint256) {
13        return _value;
14    }
15 }

```

Listing 6.1: Implementation of the counter mechanism in a simple storage contract.

State Mutability In Solidity, the state mutability of a function explicitly declares its behavior regarding state. Functions can be labeled as `view` or `pure`, promising not to modify state. Because updating the counter changes persistent storage, incrementing it is inherently a state-modifying operation. Therefore, the counter must never be incremented in functions declared as `view` or `pure`.

Implementation Example The implementation of the interaction counter mechanism is straightforward. We declare an unsigned integer state variable to hold the interaction counter and increment it at the beginning of every relevant function. We demonstrate this using an example: a simple storage contract that allows users to store a single unsigned integer value and later retrieve it. Listing 6.1 shows the implementation.

In this example, the `store` function is marked as `external` since it is designed to be called by users and other contracts. Whenever a value is stored, the counter is incremented, ensuring that every state-modifying transaction is recorded. The `receive` function, by contrast, is declared as `view` because it does not alter the state and thus does not affect the counter. This pattern generalizes to larger, more complex smart contracts.

6.1.2 Hash Chain Mechanism Implementation

This section details the implementation of the hash chain mechanism into Solidity smart contracts. While the interaction counter (Section 6.1.1) enables verifiable completeness by counting state-modifying function invocations, the hash chain enables verifiable integrity and completeness over the sequence of emitted events. Its goal is to generate a rolling cryptographic commitment for every event, resulting in a tamper-evident log that allows sparse nodes to prove no events have been omitted, reordered or altered.

EVM Logging and Solidity Events The implementation builds upon Solidity’s event system, which is an abstraction on top of the EVM’s native LOG0 to LOG4 logging opcodes [29]. When an event is emitted in Solidity, the EVM records a log entry in the transaction receipt containing (i) the emitting contract’s address, (ii) up to four topics, and (iii) arbitrary-length binary data. The semantic meaning of these logs is defined externally by the contract’s ABI, which specifies how the binary data should be interpreted. This separation makes the hashing process non-trivial because the raw log data alone is insufficient without a consistent encoding strategy.

It is important to note that our implementation does not support *anonymous events*. In Solidity, the hash of the event signature is normally included as one of the log topics, thereby enabling efficient filtering for specific events by name. For anonymous events, however, this signature hash is omitted [51]. While anonymous events offer advantages such as reduced deployment and execution costs and the ability to declare four indexed arguments rather than three, these benefits come at the cost of reduced verifiability and filterability. Since our implementation relies on the event signature both to uniquely identify and to correctly decode events—particularly in cases where a contract emits multiple different event types—anonymous events are excluded from the current design. In practice, however, anonymous events are rarely used, meaning that this restriction does not significantly limit applicability.

The Need for a Canonical Encoding To correctly commit event to the hash chain, each event’s data must be converted into a unique, deterministic byte representation. This requires a *canonical encoding* of the event’s semantic content before computing its hash. Formally, the encoding must be a bijective function: distinct events must always produce distinct encodings, and the encoding must be fully reversible to recover the original event data.

Definition 17 (Canonical Event Encoding). Let e denote an event. The canonical encoding f_{enc} of e is a deterministic function

$$f_{\text{enc}} : \mathcal{E} \rightarrow \mathcal{E}_{\text{enc}} \quad (6.1)$$

that maps each event $e \in \mathcal{E}$ to a unique representation $e_{\text{enc}} \in \mathcal{E}_{\text{enc}}$. The encoding function is bijective, meaning:

- (i) For any two events $e_1, e_2 \in \mathcal{E}$, if $f_{\text{enc}}(e_1) = f_{\text{enc}}(e_2)$, then $e_1 = e_2$, which ensures that no two distinct events share the same encoded representation.
- (ii) There exists an inverse function $f_{\text{enc}}^{-1} : \mathcal{E}_{\text{enc}} \rightarrow \mathcal{E}$, such that for every encoded event e_{enc} , we have $f_{\text{enc}}^{-1}(e_{\text{enc}}) = e$, which ensures that the encoding preserves all information about the event.

In practice, Solidity’s `abi.encode` function serves as this canonical encoding f_{enc} , providing a standard, deterministic byte representation of event parameters.

```

1 contract Storage {
2
3     event StorageUpdate(address indexed sender, uint256 val);
4
5     bytes32 private _head;
6
7     uint256 private _value;
8
9     function store(uint256 val) external {
10         _head = keccak256(abi.encode(head, msg.sender, val));
11         _value = val;
12
13         emit StorageUpdate(msg.sender, val);
14     }

```

Listing 6.2: Implementation of the hash chain mechanism in a simple storage contract.

Core Implementation Pattern The on-chain implementation of the hash chain requires three core components: *(i)* a variable to persist the current hash chain head, *(ii)* a function to compute the new head by hashing the previous head with the new event data, and *(iii)* the emission of the application event.

Unlike the interaction counter, which must be incremented at least once per transaction, the hash chain must be updated once per emitted event, since every event must be cryptographically linked.

Implementation Example Listing 6.2 demonstrates this pattern, integrating the hash chain into the simple storage contract introduced in Section 6.1.1. Each time the `store` function is called, the contract *(i)* computes a new hash chain head by combining the previous head, the sender’s address, and the stored value, *(ii)* updates the head state, and *(iii)* emits an event describing the change. This pattern ensures that every emitted event is cryptographically chained to its predecessor. Given the final hash chain head, a sparse node can verify that the received sequence of events has not been tampered with.

6.2 Proof-of-Concept Sparse Node Implementation

To evaluate the practical feasibility of the proposed protocol, we implemented a PoC client in Go, leveraging the go-ethereum (Geth) library. Our prototype demonstrates both state-based and event-based synchronization, implementing the core functionality required to maintain a consistent view of a selective subset of the blockchain derived from a custom predicate. While the theoretical protocol design presented in Chapter 4 provides a clean abstraction, building this functionality on top the Ethereum JSON-RPC API introduced several practical challenges.

We first present the architecture of the prototype we’ve developed, outlining its key components and their interactions. We then discuss the main implementation challenges

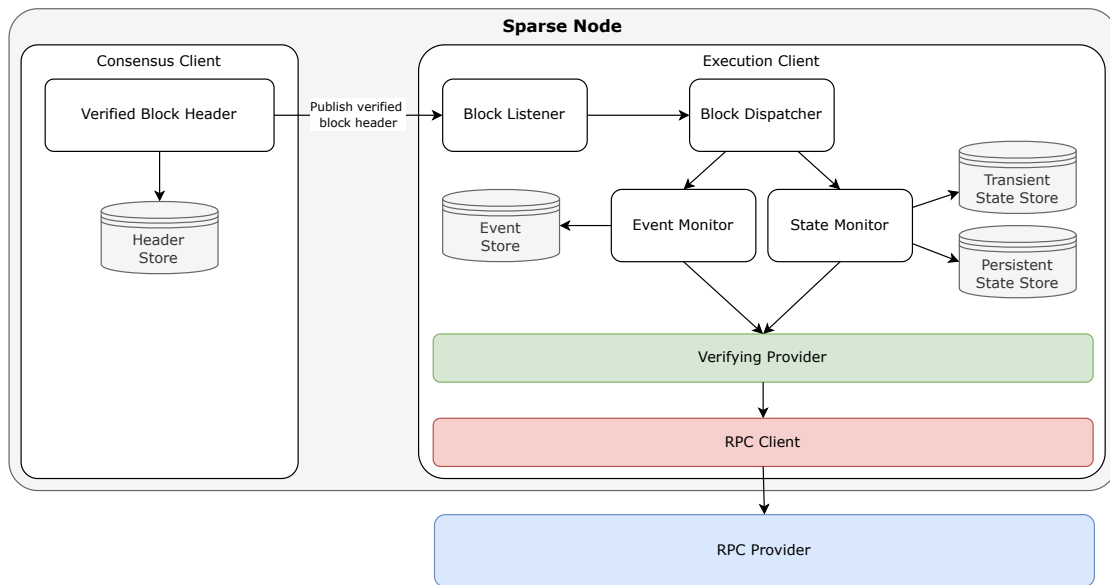


Figure 6.1: High-level architecture of the sparse node. The node consists of two main components: (i) a consensus client, responsible for tracking the canonical chain and providing verified block data, and (ii) an execution client, which processes relevant events and transactions to reconstruct the sparse event log and sparse ledger, respectively.

we encountered and the strategies we adopted to address them.

6.2.1 Architecture

We now provide a detailed overview of the sparse node architecture, highlighting its key components and their interactions. At a high level, the sparse node is composed of two primary clients: a *consensus client* and an *execution client*. The consensus client tracks the canonical chain, validates block headers, and ensures the integrity of the consensus layer. The execution client manages application-specific data: in event mode, it maintains the sparse event log, whereas in sparse mode, it maintains the sparse ledger together with its corresponding sparse state. This modular design is illustrated in Figure 6.1.

Whenever a new block header is published, the consensus client first validates it to ensure it conforms to the canonical chain rules and then persists it locally. After validation, the consensus client forwards the header to the execution client. Inside the execution client, a dedicated *block listener* receives the header and passes it to a *block dispatcher*, which distributes it to all components that need it. Depending on the mode of operation, these components include:

- Multiple *event monitors* (in event mode), each dedicated to tracking and processing the events emitted by a specific smart contract. Each monitor independently

retrieves relevant events from the blocks it receives, verifies their integrity using the event hash chain, and stores them in the local event log for subsequent use.

- A single *state monitor* (in sparse mode), responsible for processing all transactions relevant to the tracked accounts. This monitor executes transactions against the sparse state, maintains the sparse ledger, and ensures consistency and correctness of the reconstructed state across all monitored accounts.

In practice, the underlying blockchain data is typically obtained from an untrusted *RPC provider*. To mitigate this trust assumption, the execution client wraps the RPC interface with a *verifying provider*, which leverages cryptographic proofs to validate data whenever possible (e.g., verifying storage values against inclusion proofs).

Event Monitor In event mode, the execution client instantiates one or more event monitors. Each monitor is bound to a specific smart contract and configured with the corresponding event predicate. Its role is to maintain the sparse event log for that contract, which is persisted in the *event store*. Because event monitors are contract-specific and independent of one another, they can operate concurrently, enabling parallel event tracking across different contracts.

State Monitor In sparse mode, the execution client instantiates a single state monitor. Unlike event monitors, which operate independently, the state monitor is responsible for reconstructing and maintaining both the sparse ledger and the corresponding sparse state across all subscribed accounts. Since transaction execution may involve dependencies between accounts and different parts of the global state, centralization in one monitor is required to guarantee consistency and integrity.

For this purpose, the state monitor employs two distinct databases: a *transient state store*, used during transaction execution to temporarily hold state keys not part of the sparse state, and a *persistent state store*, which maintains only the sparse state itself. This separation ensures correct execution semantics while minimizing storage overhead.

6.2.2 Implementation Challenges

While the architectural design of the sparse node follows the clean abstractions of our protocol, implementing these abstractions on top of Ethereum’s existing JSON-RPC API introduced several non-trivial challenges. In practice, the RPC interface was not designed with sparse nodes in mind, leaving important verification primitives either unsupported or only indirectly accessible.

In the remainder of this section, we examine the most significant challenges encountered during the implementation of our prototype, together with the strategies we adopted to overcome them. These experiences highlight the gap between the protocol-level design and its realization in today’s Ethereum infrastructure, and point to concrete areas where future protocol or client support could substantially simplify sparse node implementations.

Challenge 1: Transaction Inclusion A core requirement of the Sparseth protocol is that the verifier can independently confirm the inclusion of a transaction in a block using the transaction root in the block header (see Line 20 of Algorithm 4.4). Ideally, the verifier should receive a transaction together with a cryptographic proof, enabling local validation against the header’s transaction root.

In practice, however, the Ethereum JSON-RPC interface provides no method to retrieve such proofs. RPC calls return only raw transaction data without any verifiable evidence of inclusion. As a result, the sparse node must reconstruct the proof locally by downloading all transactions in the block, rebuilding the MPT, and verifying that the computed transaction root matches the one in the block header.

Challenge 2: Identifying Relevant Transactions A second challenge arises from the need to identify all transactions in a block that affect a particular account. In the abstract protocol (see Line 5 of Algorithm 4.4), the sparse node first sends its state predicate \mathcal{X}_s to the prover. This informs the prover which subset of transactions is relevant to the verifier.

In practice, however, the Ethereum JSON-RPC interface provides no method to retrieve only those transactions that touch a specific address. Unlike events, where RPC calls can return all logs emitted by a given smart contract within a block, transaction queries return only the raw list of transactions without any filtering capabilities. Consequently, the sparse node must download the complete set of transactions for each block and apply the filtering function ϕ locally to determine which transactions satisfy its state predicate. This trade-off incurs an additional bandwidth and computational overhead.

Challenge 3: Obtaining the Transaction Read Set A third challenge arises from the need to determine the read set of each transaction, i.e., all accounts and storage locations accessed during execution, which may not be part of the sparse state (see Line 12 of Algorithm 4.4). This information is essential for the sparse node to correctly update its sparse state and ensure consistency with the global state.

In our implementation, we leverage the `debug_traceTransaction` RPC method together with a custom *prestate tracer* to extract all touched accounts and storage slots for a given transaction. However, this approach has two practical limitations. First, the `debug_traceTransaction` endpoint is not universally supported by all RPC providers, as it resides in the `debug` namespace rather than the standard `eth` namespace. Second, the read set itself cannot be cryptographically verified: no inclusion or integrity proof exists for accessed accounts or storage slots. To address this, the sparse node monitors the transient state store during transaction execution, ensuring that no other accounts or storage slots outside the expected read set are accessed.

Despite these challenges, our implementation demonstrates that sparse nodes can be constructed today using current infrastructure, providing verification of both event logs and sparse state. Moreover, the experiences gained highlight concrete areas where

improvements to client APIs or protocol support—such as native inclusion proofs and address-filtered transaction queries—could significantly reduce overhead and simplify deployment of sparse nodes in practice.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Evaluation

This chapter evaluates the practical costs and benefits of the proposed sparse-node protocols in realistic deployment settings. Our goal is to quantify their resource footprint along three complementary dimensions: on-chain gas overhead, client-side resource usage, and operational costs.

We begin by measuring the additional gas overhead introduced by the interaction counter and the hash chain commitment. These measurements isolate the recurring per-call and per-event costs, which we then compare against representative dApp transactions such as Tether transfers, CryptoKitties auctions, and Uniswap V2 swaps.

Next, we benchmark our PoC client in both event mode and sparse mode. Here, we evaluate bandwidth consumption, persistent storage requirements, and computational overhead under a workload calibrated to Uniswap V2 activity. The results highlight that event mode achieves drastic reductions in bandwidth and storage by downloading only headers and relevant logs, while sparse mode preserves strong storage and compute benefits but currently suffers from elevated bandwidth usage due to API limitations in transaction filtering.

Finally, we translate these resource profiles into monthly operational costs under AWS pricing. This comparison demonstrates substantial cost savings relative to full and archive nodes, while also outlining the conditions under which each sparse-node mode offers the greatest economic advantage.

7.1 Gas Cost Overhead

The Sparseth and Eventeth protocols proposed in Chapter 4 require contracts to integrate specific verification primitives—namely the counter mechanism and the hash chain mechanism. The primitives are formally described in Section 4.3 (counter) and Section 4.2 (hash chain), and their practical Solidity implementations are detailed in Section 6.1.1

and Section 6.1.2, respectively. While essential for providing sparse validity and verifiable completeness, these primitives introduce additional computation and storage overhead on-chain, which increases gas consumption per transaction.

A critical consideration is that these additional costs are borne by the users submitting transactions, not by the dApp operators running the sparse nodes who benefit from the sparse node infrastructure. It is important to note, however, that monitoring EOAs incurs zero gas overhead. Given low transaction fees are a critical factor for user adoption and the long-term viability of dApps, a precise quantification of the gas overhead imposed by our approach is essential. Furthermore, we discuss several mitigation strategies to reduce this overhead.

A precise analysis of the gas overhead requires an understanding of the Ethereum storage model, particularly the distinction between *cold* and *warm* storage accesses introduced in EIP-2929 [52]. Accessing a storage slot for the first time in a transaction is a *cold access* and incurs a higher cost, reflecting the initial disk I/O. Subsequent accesses to the same slot within the transaction are warm accesses and are significantly cheaper [52].

Example The first time a contract reads a storage variable in a transaction (SLOAD) is a cold access costing 2,100 gas. Any subsequent read of that same variable within the transaction is a warm access, costing only 100 gas [52].

This section presents a detailed analysis of the gas cost overhead introduced by our protocol. We provide a granular breakdown of the costs associated with:

- Maintaining and updating the interaction counter.
- Constructing and storing the hash chain head commitments.

The analysis evaluates this overhead in the context of warm and cold access patterns and assesses its relative impact in typical transaction costs of popular dApps hosted on the Ethereum platform. Our measurements were conducted by deploying Solidity smart contracts, compiled using the Solidity compiler (v0.8.20) on a local Anvil testnet (v1.2.3-stable), managed through the Remix IDE (v0.69.0).

7.1.1 Cost of Counter Maintenance

The counter mechanism requires a smart contract to maintain a single storage variable that is incremented for each state-modifying function. The gas cost is dominated by storage operations (SLOAD and SSTORE) whose pricing depends on access patterns as discussed in the previous section. While arithmetic (e.g., ADD) and stack operations (e.g., PUSH) are required, their cumulative cost—typically 3 gas per operation—is orders of magnitude smaller than the significant expense of reading and writing contract storage [29].

The process of incrementing the counter involves two key storage operations within the function call:

Instruction	Gas Cost	Description
SLOAD (cold)	2,100	Load cold word from storage
SSTORE (update)	2,900	Save warm word to storage
Other	154	Arithmetic operations and stack management
Total Overhead	5,154	

Table 7.1: Gas cost breakdown for maintaining an interaction counter per function call.

1. An SLOAD operation to read the current value of the counter from storage. The cost of this operation is determined by whether the storage slot was already accessed within the current transaction or not: 2,100 gas for a cold access or 100 gas for a warm access.
2. A subsequent SSTORE operation to write the new, incremented value back to the same storage slot. As this slot has already been accessed, this operation is priced as a warm update, costing 2,900 gas.

It is important to distinguish this recurring update cost from the one-time initialization cost. The first transaction that initializes the counter (transitioning it from zero to a non-zero value) incurs a one-time warm SSTORE cost of approximately 22,000 gas. Given the counter's strictly monotonic nature, this initialization occurs only once over the entire lifetime of the contract. To optimize future interactions, the party deploying the contract may absorb this initial cost by pre-initializing the counter in the constructor, ensuring all subsequent updates incur only the lower recurring gas cost.

The recurring per-function overhead of the counter mechanism after initialization was empirically measured by deploying two contract variants: One implementing the counter mechanism and one without it. The same function was invoked with identical inputs on both variants in isolated transactions that executed exactly this one function. The difference in overall transaction gas costs between the two variants was then calculated, thereby isolating the net overhead of maintaining the interaction counter. Under a worst-case scenario where the read operation is cold, the measured overhead amounts to 5,154 gas, as summarized in Table 7.1.

The economic burden of this additional gas consumption is shifted to the users who interact with the smart contract. At an average gas price of 0,728 Gwei and an Ether price of USD 4,571 (as of 2025-08-26), the resulting overhead amounts to approximately USD 0,02 per state-modifying function call.

To contextualize the practical impact of the counter mechanism, we compare it against the gas costs of some prominent Ethereum dApp transactions. We analyze three actions: (a) a stablecoin transfer (Tether USDT), (b) a collectible market operation (CryptoKitties

auction creation), and (c) a DeFi swap (Uniswap V2). The average gas costs for the analyzed actions were obtained from Etherscan [53].

- (a) **Tether:** Tether is one of the most widely used ERC-20 tokens on Ethereum. A token transfer is executed through the `transfer` function, which updates the balances of the sender and recipient in contract storage and emits a `Transfer` event [54]. The typical gas cost for this action is approximately 46,000 gas. The additional 5,100 gas overhead introduced by the sparse node-compatible interaction counter increases the total gas cost by about 11%.
- (b) **CryptoKitties:** CryptoKitties is a NFT collectibles game, built on top of the ERC-721 token standard [55]. Unlike a simple token contract, the application logic is distributed across a system of four contracts. Creating an auction sale on the platform involves interactions with two of these core contracts. Consequently, the total overhead for this action is $2 \times 5,100$ gas. Given that the typical gas cost for this operations of approximately 132,000 gas, the additional 10,200 gas overhead increases the total gas cost by 7%.
- (c) **Uniswap:** Uniswap V2 is a foundational protocol for automated ERC-20 token exchange, powering the Uniswap DEX [56]. Its architecture separates responsibilities between core contracts (the immutable pair contracts for each token pair) and periphery contracts (a router, which provides a user-friendly interface). A typical token swap initiated via a router contract involves the following path:
 - a) The user calls the router contract.
 - b) The router calls the specific pair contract for the token swap.

In practice, swapping one ERC-20 token for another through a router typically involves interactions with four contracts in total. The average gas cost for such a swap is approximately 125,000 gas. Introducing an additional overhead of 20,400 gas increases the total gas cost by roughly 16%. However, if the protocol does not monitor the individual token contracts (which are not directly part of the Uniswap application), the overhead can be reduced to about 8%. Furthermore, due to Uniswap's permissionless design, the router contract has no specific privileges, and gas consumption can be potentially be optimized further.

A comparison of the gas overhead introduced by the sparse node-compatible interaction counter across these representative applications is summarized in Table 7.2. The results show that the relative impact of the protocol depends strongly on the complexity of the underlying transaction and number of involved contracts.

7.1.2 Cost of Hash Chain Commitments

The hash chain mechanism stores a rolling commitment to all events emitted by the contract, enabling verifiable event logs. This requires a smart contract to maintain a

Application	Base Gas	Overhead Gas	Relative Increase
Tether (ERC-20 Transfer)	46,000	5,100	$\approx 11\%$
CryptoKitties (Auction Creation)	132,000	10,200	$\approx 7\%$
Uniswap V2 (Token Swap)	125,000	10,200 – 20,400	$\approx 8\% - 16\%$

Table 7.2: Comparison of the additional gas overhead introduced by the sparse node-compatible interaction counter across different popular Ethereum dApps. For Uniswap V2 token swaps, the reported range reflects two scenarios: The upper bound corresponds to monitoring all four involved contracts. The lower bound applies when the protocol does not monitor the individual ERC-20 token contracts, which are not part of the core Uniswap application.

single storage variable (the head of the hash chain) that is updated for each function which emits an event. The gas cost overhead arises from (i) reading the current head from storage, (ii) computing the new head from the previous head and the current event data, and (iii) writing the updated head back to storage.

A critical distinction from the counter approach (discussed in Section 7.1.1) is that the computational cost of the hash chain is variable rather than fixed. Specifically, the cost of the KECCAK256 to compute the new head chain commitment is dependent on the size of its input, which includes the entire event data. The base cost for this operation is 30 gas, with an additional cost of 6 gas for each word of input [29]. Therefore, the total gas cost overhead of the hash chain mechanism is a function of both fixed storage costs and variable computational costs tied to the application’s event schema.

To quantify this cost in practice, we empirically measured the recurring per-event overhead of maintaining the hash chain head using the same methodology as for the counter measurements. For an event schema containing the message sender and an unsigned integer (32 bytes), the observed overhead amounts to 5,527 gas, as summarized in Table 7.3. Similar to the counter mechanism, the gas consumption is primarily dominated by storage operations, with the computational cost of hashing contributing only marginally.

At an average gas price of 0,728 Gwei and an Ether price of USD 4,571 (as of 2025-08-26), this corresponds to an economic overhead of USD 0,02 per event emission.

To contextualize the practical impact of the hash chain mechanism, we compare its additional cost against the gas cost of the same three representative dApp interactions analyzed in Section 7.1.1: (a) an ERC-20 transfer of Tether, (b) the creation of an auction on CryptoKitties, and (c) a Uniswap V2 token swap.

- (a) **Tether:** The transfer function emits a single `Transfer` event, which records that value tokens were moved from one account (`from`) to another account (`to`). Since

Instruction	Gas Cost	Description
SLOAD (cold)	2,100	Load cold word from storage
SSTORE (update)	2,900	Save warm word to storage
KECCAK256	48	Compute Keccak-256 hash
Other	479	Arithmetic operations and stack management
Total Overhead	5,527	

Table 7.3: Gas cost breakdown for maintaining a hash chain commitment per event emission. The hashing cost is an estimate for a typical event data size.

the event schema is small, consisting of just the sender, recipient, and value, the additional gas cost overhead for maintaining the hash chain head is 5,681 gas. Relative to the typical ERC-20 transfer cost of approximately 46,000 gas, this corresponds to an increase of approximately 12%.

- (b) **CryptoKitties:** When creating an auction for a cat, the contract emits a single `CreateAuction` event. This event records the identifier of the cat being listed, the starting and ending prices, and the duration of the auction. All these parameters are represented as 32 bytes unsigned integers, the additional gas cost for maintaining the hash chain head amounts to 5,695 gas. Compared to the typical gas cost for creating an auction, this corresponds to a relative increase of 4%.
- (c) **Uniswap:** During a token swap, the contracts emit one `Swap` event in the pair contract and two `Transfer` events—one for each of the ERC-20 token contracts involved. The `Swap` event includes six parameters, detailing the sender, recipient, and the amounts of tokens exchanged. Including the two `Transfer` events, the addition gas cost for maintaining the hash chain head amounts to $6,005 + 2 \times 5,681 = 17,367$ gas, corresponding to an overhead of approximately 13%. However, if the protocol does not monitor the events emitted by the individual token contracts, the overhead can be reduced to roughly 5%.

A summary of the gas overhead introduced by the sparse node-compatible hash chain mechanism across these representative applications is provided in Table 7.4. The results indicate that the relative impact of the protocol depends strongly on both the complexity of the underlying transaction and the number of events involved.

7.1.3 Mitigation Strategies

The additional gas costs incurred by sparse-node-compatible contracts can be mitigated through a combination of technical optimizations and economic mechanisms. While

Application	Base Gas	Overhead Gas	Relative Increase
Tether (ERC-20 Transfer)	46,000	5,681	$\approx 12\%$
CryptoKitties (Auction Creation)	132,000	5,695	$\approx 4\%$
Uniswap V2 (Token Swap)	125,000	6,005 – 17,367	$\approx 5\% - 13\%$

Table 7.4: Comparison of the additional gas overhead introduced by the sparse node-compatible hash chain mechanism across different popular Ethereum dApps. For Uniswap V2 token swaps, the reported range reflects two scenarios: The upper bound corresponds to monitoring all three involved events. The lower bound applies when the protocol does not monitor the individual ERC-20 token contracts, which are not part of the core Uniswap application.

several approaches are outlined here, a detailed exploration of mitigation strategies lies outside the scope of this thesis and is left for future work.

A particularly effective technical measure is to deploy these applications on Layer-2 (L2) rollups (e.g., Arbitrum or Optimism). Base gas fees on such networks are orders of magnitude lower than on Ethereum Mainnet, rendering the overhead from sparse node commitments negligible [1]. For this architecture to work, the entire verification stack—including the smart contracts implementing the counter or hash-chain mechanisms and the verifier-client logic—must reside on the same L2 network. Our protocols are natively compatible with EVM-based rollups (e.g., Arbitrum or Optimism) because they rely on the EVM for transaction execution and produce a rollup ledger. A sparse node then synchronizes with and verifies the state of that specific chain, benefiting from its low transaction fees while remaining within its security model [57].

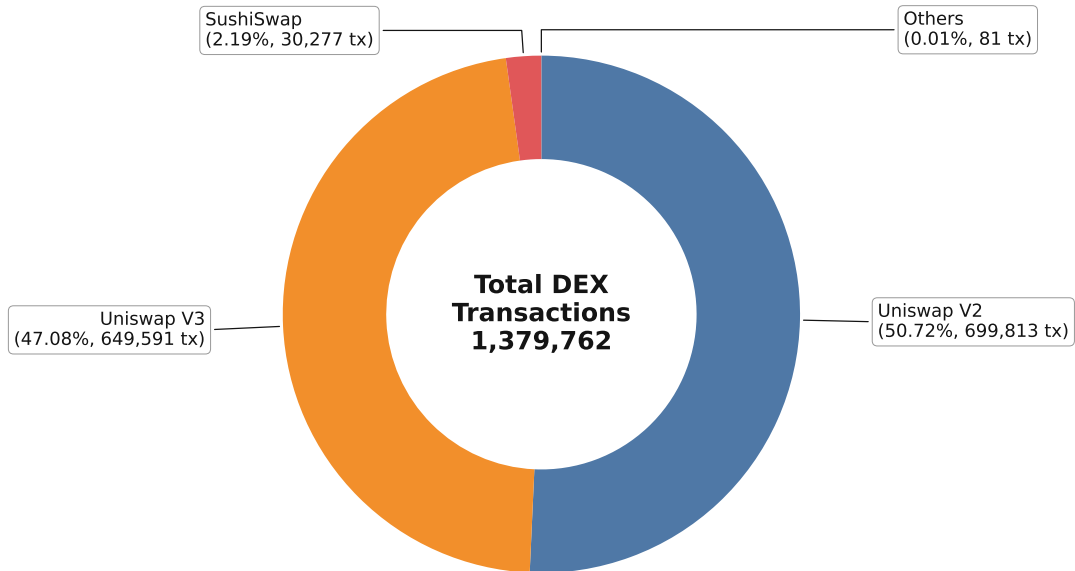
Beyond technical deployment, economic models can be designed to redistribute or absorb the additional costs, aligning incentives between users and dApp operators who benefit from the infrastructure. One such approach is the implementation of a rebate and reward pool. This approach involves a smart contract-managed pool, funded by a small fraction of the application’s revenue. Users who submit transactions that update the sparse node’s on-chain commitment become eligible to claim a partial or full rebate from this pool, effectively reimbursing them for the auxiliary gas cost incurred.

Collectively, these mitigation strategies can significantly reduce the economic impact of sparse node operation, making it a practical option for a wide range of dApps. We plan to explore these further as a future work.

7.2 Benchmarks

This section presents a comparative performance analysis of the resource consumption of the sparse client—evaluated in both sparse and event mode—relative to an Ethereum full node. The objective of this evaluation is to quantify the efficiency gains achieved by

Ethereum Mainnet DEX Market Share (09/14/2025 - 09/21/2025)



Source: Etherscan.io

Figure 7.1: Market share distribution of DEX platforms on Ethereum Mainnet by transaction volume (September 14–21, 2025). Uniswap V2 dominated with 50,72% (699,813 transactions), followed by Uniswap V3 (47,08%, 649,591 transactions), SushiSwap (2,19%, 30,277 transactions), and other DEXs (0,01%, 81 transactions). Total DEX transactions during this period amounted to 1,379,762. (Source: Etherscan)

our protocols by assessing the reduction in bandwidth usage, computational overhead, and storage requirements under a realistic workload.

7.2.1 Workload Selection

To establish a representative benchmark, we began by analyzing Ethereum Mainnet activity to identify dominant dApp patterns. We examined a seven-day period from September 14 to September 21, 2025 to determine the most actively used Decentralized Exchange (DEX). Figure 7.1 illustrates the market distribution among leading DEX platforms during this interval. The analysis revealed Uniswap V2 as the predominant DEX, processing 50,72% of all DEX transactions with a daily average of 99,973 transactions.

To contextualize this exchange activity within broader network patterns, we conducted a complementary one-year analysis of Mainnet transactions from September 1, 2024 to September 1, 2025. Figure 7.2 displays the daily transaction counts, showing a median of 1,274,439 transactions per day. This robust measure was selected over the arithmetic mean

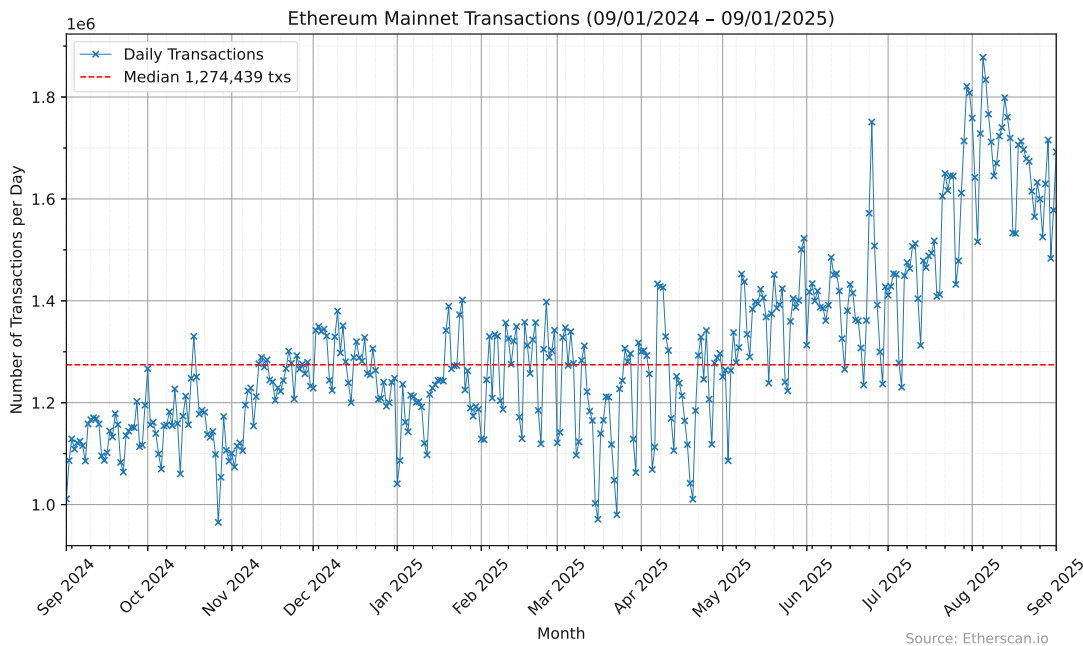


Figure 7.2: Daily transaction volume on Ethereum Mainnet (September 1, 2024 to September 1, 2025). The network maintained a median daily transaction count of 1,27 million transactions. (Source: Etherscan)

to mitigate the influence of short-term volatility. Within this total volume, Uniswap V2 transactions constituted approximately 7,84% of daily network activity. This substantial market share establishes Uniswap V2 as an ideal stress-test benchmark, representing a demanding edge-case scenario for resource monitoring. In contrast, most other dApps would impose substantially lower operational loads.

7.2.2 Simulation Contracts

To generate the calibrated workload, we implemented and deployed a suite of smart contracts that replicate the core exchange mechanics of Uniswap V2. The simulation comprises three primary components deployed on Sepolia testnet: two ERC-20 tokens representing trading assets and a simplified exchange contract that handles token swaps.

- Quark ERC-20 Token `0xb892728166b5e78733eCf42891276BaDcd95E745`
- Photon ERC-20 Token `0x5C9F5a0b679d47AC760127a82A2C1534D197c9B8`
- Exchange Contract `0x19f080cfc49f6f36fee8ca23bf7d78819887a56b`

ERC-20 tokens implement the standard interface using OpenZeppelin's audited templates, serving as the fundamental trading assets. The exchange contract emulates a Uniswap

V2 Pair contract, capturing the essential economic and computational characteristics of the swap functionality while maintaining a simplified architecture. Crucially, our simulation preserves the dependency patterns found in Uniswap V2, where the exchange contract interacts with external ERC-20 token contracts. This design ensures that our benchmark accurately reflects the inter-contract communication patterns that impact node performance, even though our implementation omits some computational complexities of the full protocol.

7.2.3 Workload Simulation

With the contract infrastructure in place, we developed a scaling methodology to recreate Uniswap V2's transactional load in a controlled test environment. This approach ensures that our simulated workload accurately reflects real-world operational conditions relative to baseline network capacity.

We first established a baseline by analyzing Sepolia testnet activity during the identical one-year period (September 1, 2024 to September 1, 2025). As shown in Figure 7.3, Sepolia exhibited a median daily transaction volume of 951,862 transactions. Applying the 7,84% proportion derived from Mainnet analysis yielded our target simulation load:

$$951,862 \times 0,0784 \approx 74,625 \text{ transactions per day} \quad (7.1)$$

This represents the number of simulated Uniswap V2 transactions required to maintain equivalent proportional load conditions on the testnet. For practical benchmarking, we applied the scaled daily workload over a one-hour period. The target hourly transaction rate was calculated by distributing the daily workload evenly across 24 hours:

$$\frac{74,625}{24} \approx 3110 \text{ transactions per hour}$$

This approach maintains the average transaction density of a full day within a practical one-hour testing window.

We deployed and executed 3,190 transactions over one-hour period on the Sepolia testnet, achieving 103% of our target transaction rate. These transactions interacted with both swap functions of our exchange contract, simulating simple trading patterns. However, due to elevated background network activity during the test period, the relative load represented 6,54% of total network transactions, slightly below our target proportion of 7,84%. Figure 7.4 illustrates the distribution of our simulated transactions against the total network activity.

7.2.4 Experimental Setup

The benchmarking environment consisted of two Ethereum node configurations deployed on the same Virtual Machine (VM) running Ubuntu 24.04.1 LTS. The baseline full node was configured in archive mode and fully synchronized from the genesis block, using the Prysm consensus client (v6.0.4) together with the Geth execution client (v1.16.2-stable).

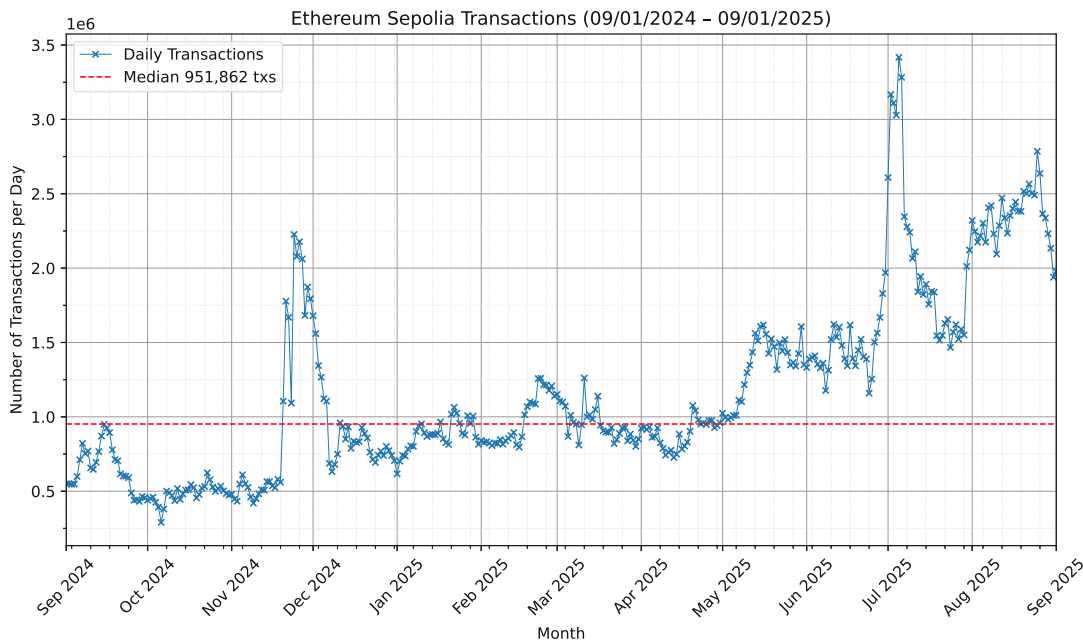


Figure 7.3: Daily transaction volume on Ethereum Sepolia testnet (September 1, 2024 to September 1, 2025). The network sustained a median daily transaction count of 0,95 million transactions. Data were collected via our deployed full node by querying each block’s transaction count using the `eth_getBlockTransactionCountByNumber` JSON-RPC endpoint and aggregating results daily.

This setup represents a complete historical node capable of serving any past state data. The sparse client was deployed alongside the full node on the same VM and evaluated in two separate configurations: event mode and sparse mode, allowing a comparative analysis of resource consumption under both operating modes. Both sparse client configurations were configured to monitor the exchange contract and the two token contracts, ensuring comprehensive coverage of all relevant state changes and event emissions.

All smart contract operations—including deployment and transaction generation—were performed using the Foundry development suite. In particular, Forge was used for contract compilation, testing, and deployment, targeting an average rate of approximately 3,110 transactions per hour.

7.2.5 Resource Consumption Analysis

With the workload simulation and experimental setup established, this section now a comparative evaluation of resource utilization across three node architectures: a conventional full node operating in archive mode, and our sparse client implementation in both event mode and sparse mode. The analysis examines three resource dimensions

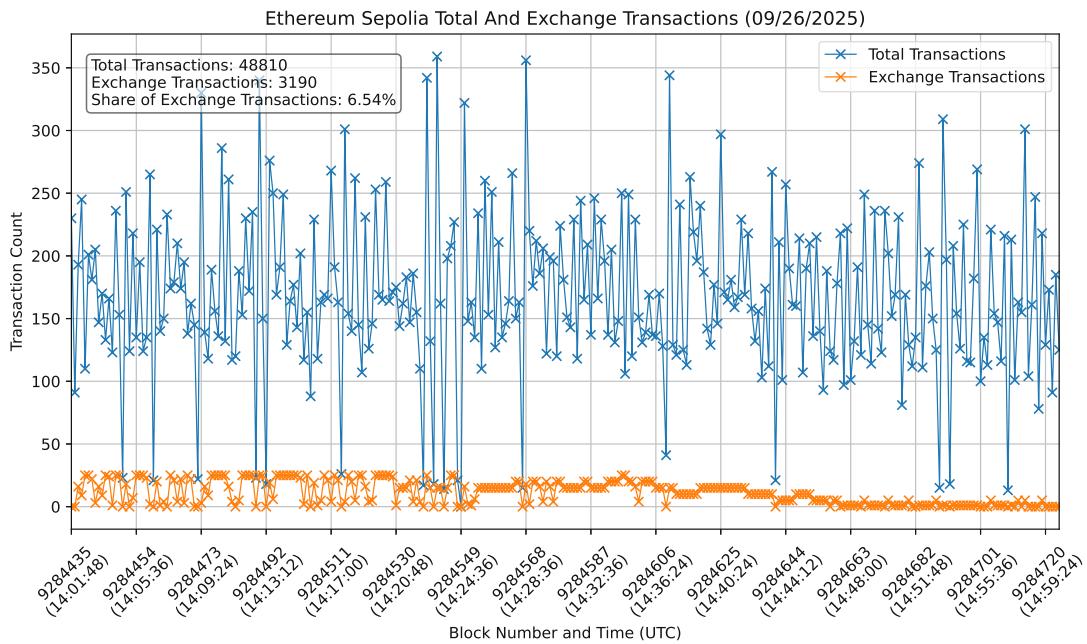


Figure 7.4: Transaction distribution on Sepolia testnet during the benchmark period (blocks 9,284,435 to 9,284,720). The blue series shows total network transactions, while the orange series represents transactions interacting with our exchange contract. Our simulation generated 3,190 transactions over 58 minutes, achieving 103% of the target rate and 6,54% of total network transactions.

(bandwidth, storage, and computation) to quantify the efficiency gains achievable through selective data processing strategies.

Bandwidth Consumption

Bandwidth consumption is often the decisive factor in whether a blockchain client can run on everyday hardware such as a server, desktop computer, or even a mobile phone. Our analysis reveals distinct patterns across the three configurations: event mode demonstrates exceptional efficiency, making it practical for end-consumer devices, whereas sparse mode exposes current API limitations.

Event Node The sparse client in event mode achieves a 95,22% reduction in bandwidth consumption compared to the full node (1,61 MB versus 33,77 MB), as shown in Figure 7.5. For this comparison, the full node’s bandwidth is estimated under a simplified model that accounts solely for block downloads, excluding the additional overhead of e.g., transaction gossiping on the Peer-to-Peer (P2P) network. Consequently, the reported 33,77 MB represents a lower bound, and the bandwidth savings achieved by event mode are likely even greater. This efficiency comes from a selective data retrieval strategy

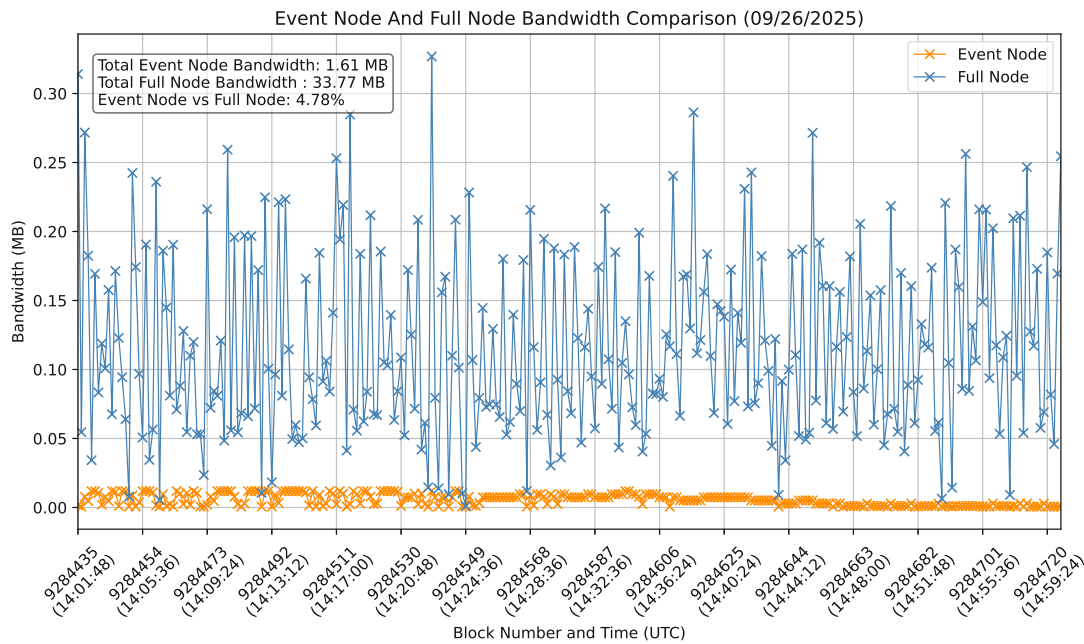


Figure 7.5: Bandwidth consumption comparison during benchmark period shows the event node (orange series) achieving 95,22% reduction (1,61 MB) compared to the full node (33,77 MB, blue series). Its efficiency stems from selective data retrieval: downloading only block headers for synchronization, event logs from monitored contracts, and hash chain commitments for event log verification. In contrast, the full node downloads complete block data, resulting in significantly higher bandwidth utilization.

that downloads only three components: *(i)* block headers for blockchain synchronization *(ii)* events emitted by monitored contracts, and *(iii)* hash chain commitments for event log verification. In contrast, the full node must download complete block data to maintain global state consistency.

Sparse Node Sparse nodes can achieve substantial bandwidth savings by downloading only those transactions that interact with monitored contracts. For example, monitoring Uniswap V2 (which represents roughly 8% of Mainnet activity) reduces bandwidth requirements by about 92% relative to a full node.

Our proof-of-concept implementation, however, highlights an important limitation of the current Ethereum infrastructure. As shown in Figure 7.6, the sparse client in sparse mode consumed 50,91 MB during the benchmark period. This is about 50,8% more than the 33,77 MB consumed by a full node. This stems from a gap in Ethereum’s JSON-RPC API: there is currently no endpoint to filter transactions by contract interaction at the block level. Consequently, our PoC client needs to fetch all transactions and their read sets, only to locally identify which ones involve the monitored contracts. With protocol

7. EVALUATION

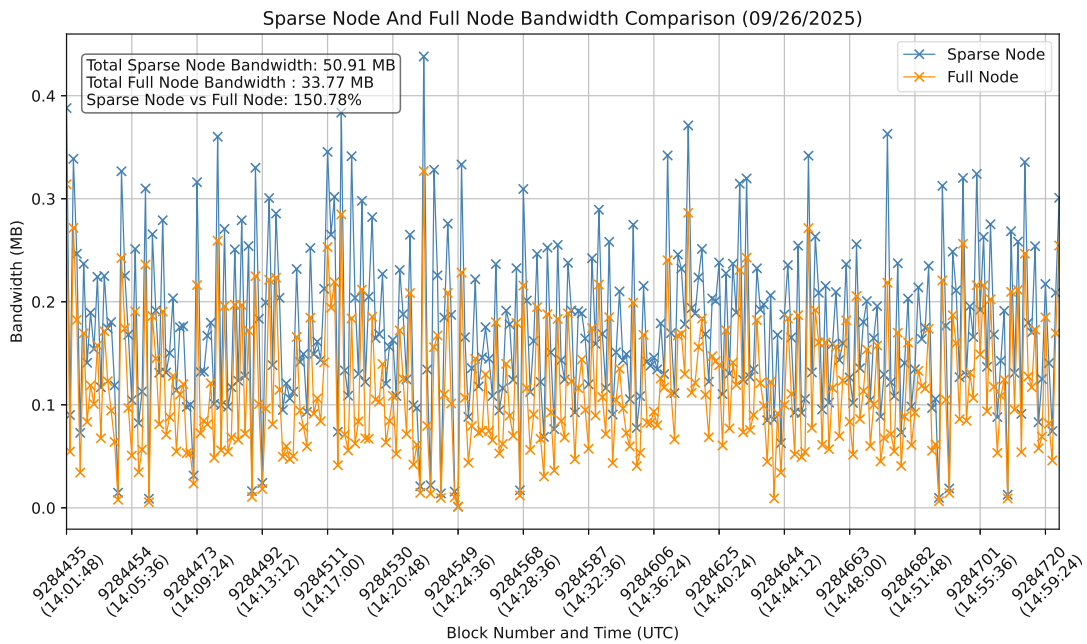


Figure 7.6: Bandwidth comparison between sparse node (blue series) and full node (orange series) during benchmark period (blocks 9,284,435 to 9,284,720). The sparse node consumed 50,91 MB (150,78% of the full node bandwidth) due to the need to download all transactions and their associated read sets for client-side filtering as well as verification proofs. In contrast, the full node—under our simplified assumption with excluded transaction gossiping—only downloads complete block data (33,77 MB). This overhead results from the absence of contract-specific transaction filtering in the current Ethereum JSON-RPC API.

enhancements—such as contract-level transaction filtering—sparse nodes can realize their theoretical efficiency gains.

Persistent Storage Size

Storage requirements demonstrate the most substantial efficiency improvements, with both sparse client configurations achieving over 99,99% reduction compared to the archive node. Rather than synchronizing from genesis, the sparse clients were initialized from a trusted checkpoint at block 9,284,435. This checkpoint was obtained from our deployed full node, though in practice such checkpoints can be sourced from various trusted providers including public APIs, infrastructure services, or verified snapshots.

This checkpoint synchronization approach is valid for our evaluation since all monitored contracts were deployed at or after this block height, ensuring complete coverage of relevant contract activity while avoiding unnecessary historical data storage.

Event Node As shown in Figure 7.7, the event node stores only 1,61 MB of data, limited to block headers for synchronization and the event logs of the monitored contracts. In our benchmark workload of 3,190 DEX transactions, the event node captured 9,576 events. This 3:1 ratio of events to transactions arises because each swap operation emits three events: two **Transfer** events from the underlying ERC-20 token contracts tracking asset movements and a **Swap** event from the exchange contract recording the trade details. This sparse event log strategy preserves complete auditability of contract events while keeping long-term storage growth minimal.

Sparse Node Sparse mode requires 2,90 MB— orders of magnitude less than the archive node’s 880 GB—by storing block headers, relevant transactions (maintaining a sparse ledger), and associated sparse state. In future iterations of our PoC implementation, storage efficiency could be further optimized by retaining only block headers that contain either handover messages from the sync committee or relevant transactions.

The storage characteristics of both sparse client configurations make them ideally suited for resource-constrained environments and long-term monitoring scenarios where archive node storage requirements become prohibitive.

Computation

Figure 7.8 shows that the sparse node executes only 3,571 transactions (7,32%) compared to the full node’s 48,810 executions, achieving a 92,68% reduction in computational workload. Specifically, a transaction is executed by the sparse node if it meets any of the following criteria:

- **Account Interaction:** The transactions touches the monitored account.
- **Contract Creation:** The transaction creates new contracts. As the created address is determined during transaction execution, it may become relevant for future monitoring.
- **State-dependent Interaction:** The transaction involves non-monitored accounts that subsequently interact with a monitored account within the same block. Since prior state changes can affect the outcome of the monitored interaction and the current Ethereum API provides state access only on block level, these transactions need to be executed as well.

The resource-consumption benchmarks demonstrate that sparse client architectures can deliver substantial efficiency gains while preserving targeted monitoring capabilities. Event mode achieves the greatest bandwidth and storage savings, reducing data transfer and persistent storage by orders of magnitude. Sparse mode, although incurring higher bandwidth usage under current infrastructure constraints, retains pronounced advantages in storage and computational efficiency. This profile makes sparse mode attractive for environments with ample network capacity but limited disk or compute resources, such

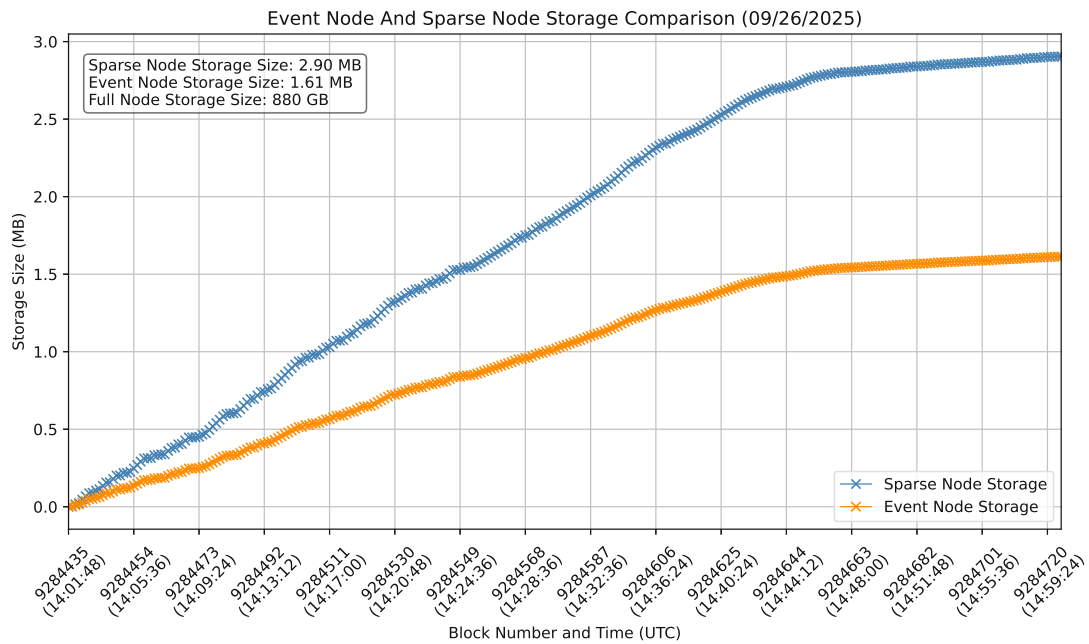


Figure 7.7: Storage size comparison between sparse client operating modes during benchmark period, synchronized from checkpoint block 9,284,435 to 9,284,720. The event node (orange series) maintains minimal storage (1,61 MB) by preserving only block headers and relevant event logs (maintaining a sparse event log). The sparse node (blue series) requires more storage (2,90 MB) as it stores block headers, re-executed relevant transactions (maintaining a sparse ledger), and associated sparse state. Both configurations demonstrate significant efficiency gains compared to the full archive node (880 GB), which stores the complete global ledger and state history since genesis.

as edge devices where storage costs dominate operational expenses. Both configurations reduce persistent storage requirements by more than 99% compared to full archive nodes, validating the feasibility of resource-efficient blockchain monitoring for application domains that demand long-term, contract-specific observation.

7.3 Operational Costs

To quantify the economic implications of the observed resource efficiencies, we performed a comparative cost assessment using current Amazon Web Services (AWS) pricing models. Monthly operational costs were derived with the official AWS pricing calculator.

For the sparse client configurations, the calculation assumes the same workload used in our benchmarks (Section 7.2)—continuous monitoring of the Uniswap V2 DEX—so that cost estimates reflect the transaction volume and data retrieval rates measured in the preceding experiments.

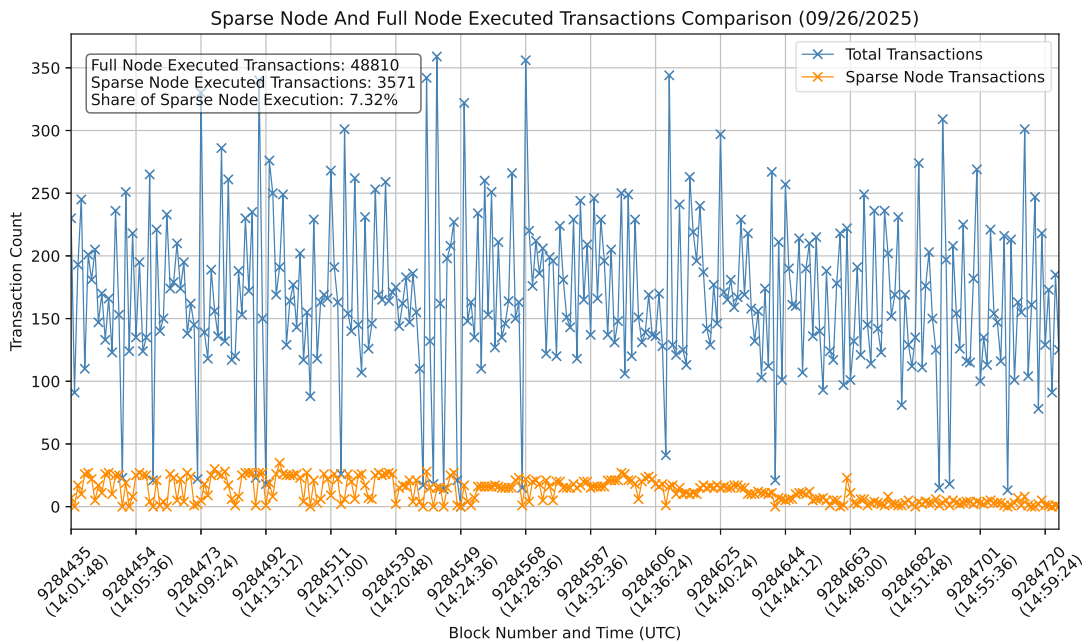


Figure 7.8: Transaction execution comparison during benchmark period shows the sparse node re-executed only 3,571 transactions (7,32% of the total 48,810 executions). The sparse node selectively re-executes transactions relevant to monitored contracts, while the full node processes all transactions to reconstruct global state. The sparse node’s execution set includes: (i) transactions interacting with monitored contracts, (ii) contract creation transactions, and (iii) transactions involving accounts that later interact with monitored contracts within the same block. This selective execution strategy achieves 92,68% reduction in computational overhead while maintaining accurate state tracking for target contracts.

This approach demonstrates how the reduced resource consumption of sparse clients translates into substantial long-term cost savings for realistic, high-activity monitoring.

7.3.1 Full Node Costs

The full node configuration follows the recommended hardware specifications from the go-ethereum (Geth) project for a pruning full node (where historical state is pruned) and a archive node (where historical state is persisted) for Mainnet deployment, including CPU, memory, and disk space. Recommended specifications include a quad-core CPU, 16 GB of memory, and 2 TB SSD for a full node or a 12 TB SSD for an archive node. For bandwidth estimation, we use the mean block size reported by Etherscan from September 1, 2024 through September 1, 2025, which is 87 KB. Given the average Ethereum block time of 12 seconds (i.e., 5 blocks per minute), this equates to about 219,000 blocks per month, or approximately 19 GB of inbound data traffic each month.

Node Type	AWS EC2	AWS EBS	Data Transfer	Monthly Costs	Rel. Cost
Full Node	t4g.xlarge	2 TB	19 GB	USD 243	100%
Archive Node	t4g.xlarge	12 TB	19 GB	USD 1,053	433%
Sparse Node	t4g.xlarge	160 GB	28,5 GB	USD 83	34%
Event Node	t4g.large	24 GB	1 GB	USD 37	15%

Table 7.5: Monthly operational cost comparison for Ethereum node configurations on AWS. Costs include EC2 instances, EBS storage, and data transfer based on Mainnet resource requirements. Percentage costs are relative to the full node baseline.

The costs for both the full node and archive node are summarized in Table 7.5.

7.3.2 Sparse Client Costs

Based on our benchmark results and AWS pricing, we project the following cost structures for Mainnet deployment of the sparse client.

Event Node Costs The sparse client in event node achieves the highest cost efficiency by minimizing CPU load, inbound bandwidth, and storage requirements through selective data retrieval. Benchmark measurements indicate that a `t4g.large` EC2 instance with 2 cores and 8 GB memory is more than sufficient for sustained operation. Bandwidth requirements are reduced to roughly 5% of those of a full node, translating to 1 GB per month compared to the full and archive node’s 19 GB. This 95% reduction in data transfer directly corresponds to storage needs, as the event node must persist only the block headers and event logs it downloads. Because the event node persists only block headers and the relevant contract event logs it downloads, long-term storage growth likewise remains near 1 GB per month. Assuming continuous operation for one year, total storage expands to only about 24 GB. As detailed in Table 7.5 these resource savings translate to an estimated operating cost of approximately USD 37 per month, an 85% reduction relative to the USD 243 monthly cost of a full node.

Sparse Node Costs The sparse client in sparse mode provides monitoring capabilities with higher resource requirements than event mode, but still achieves substantial cost savings compared to the full and archive node. For computational resources, we allocate the same `4g.xlarge` EC2 instance as the full node, since the sparse node must perform client-side transaction filtering and selective re-execution, which involves processing all transactions to identify relevant interactions. Bandwidth consumption is significantly higher than both event mode and the full node, reaching 150% of full node requirements due to the need to download complete block data, transaction read sets, and verification proofs. This translates to approximately 28,5 GB per month compared to the full node’s

19 GB baseline. For storage estimation, we apply the proportional scaling methodology used throughout our analysis. Given that Uniswap V2 transactions constitute approximately 8% of Mainnet activity, we scale the full node's 2 TB storage requirement proportionally, resulting in 160 GB of storage for sparse mode operation. As detailed in Table 7.5, the total operational cost for sparse mode is projected at USD 83 per month, representing a 66% reduction compared to the full node's USD 243 monthly cost.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.



Related Work

This chapter situates our work within the broader context of blockchain client architectures. While the concept of resource-efficient clients is not new, ranging from light clients [58] to ultralight clients [59, 60], our approach differs from these paradigms. Unlike light clients, which download block headers to identify the head of the chain and verify transaction inclusion, sparse nodes selectively re-execute transactions and reconstruct the associated state, substantially reducing resource requirements compared to a full node. The Sunfish protocol [3] represents the foundational work, introducing the concept of such clients.

Sunfish Sparse Client While our work shares the overarching goal of enabling resource-efficient verification, Sunfish suffers from critical limitations and lacks a practical implementation or thorough evaluation. Its design requires validators to explicitly include additional commitments into every block header, creating an overhead that scales linearly with the number of supported substates, hindering practical deployment on live networks. In particular, Sunfish-C requires validators to include a counter reflecting the number of transactions that touch a given sparse state of interest, while Sunfish-HC requires the construction of a per-substate hash chain over all transactions touching that state [3]. This thesis addresses these limitations by formalizing and implementing two novel protocols for EVM-compatible blockchains, that fundamentally differ in their architectural approach. Unlike Sunfish, our protocols are built entirely on execution-layer primitives and require no changes to the underlying blockchain protocol. By shifting the responsibility for commitments from validators to application-level mechanisms embedded directly in the execution layer, our design emphasizes practical deployability and validator-friendly operation, clearly distinguishing it from Sunfish.

Light Clients Light clients verify blockchain data without downloading the entire chain. They rely on block headers and cryptographic proofs (like Merkle or state root proofs)

to check that specific data is valid. This lets them operate efficiently on low-resource devices while still maintaining trust in the network’s security guarantees.

Light clients typically operate under the assumption of an honest majority of block producers. They cannot independently verify the validity of transactions or ensure that block data is available. This creates systemic risks in adversarial settings: under a dishonest majority, invalid state transitions may be accepted as valid, and withheld data can prevent the generation of fraud proofs that would otherwise expose such incorrect state. To address these vulnerabilities, two mechanisms have been proposed in [37]: compact fraud proofs for invalid state transitions, and probabilistic data availability sampling. Fraud proofs exploit an execution trace embedded in the block—consisting of intermediate state roots interleaved with transactions—alongside sparse Merkle state witnesses. This enables full nodes to generate concise proofs demonstrating that a given post-state does not correctly follow from the pre-state and the corresponding transactions. Light clients can verify these proofs locally, allowing them to reject invalid blocks without relying on an honest majority of producers. Sparse nodes differ fundamentally from light clients in that they do not rely on external fraud proofs or to detect invalid state transitions. Instead, they selectively re-execute the relevant subset of transactions and reconstruct the associated state.

While fraud proofs address correctness, ensuring the availability of block data requires complementary mechanisms. For this purpose, probabilistic sampling is employed. Block data is divided into fixed-size shares, arranged in a two-dimensional matrix, extended using 2D Reed–Solomon encoding, and committed via Merkle roots of rows and columns under a single data root. Light clients randomly sample shares and verify their inclusion through Merkle proofs. Provided that a sufficient fraction of clients participate in sampling and gossiping results, withheld data can be detected or reconstructed with high probability [37]. However, like light clients, sparse nodes remain vulnerable to data availability attacks as discussed in Section 5.4. Thus, sparse nodes could benefit directly from the same probabilistic data availability mechanisms developed for light clients.

Ethereum EIPs Beyond these protocol-level defenses, other proposals in the Ethereum ecosystem aim at reducing the operational burden of clients more generally. EIP-4444 [61] proposes limiting historical data served by Ethereum execution clients to 82,125 epochs (about a year). Under this proposal, clients may stop serving and optionally prune older block headers, block bodies or transaction receipts that fall outside this window to the P2P layer. Historical blocks and receipts occupy hundreds of gigabytes of disk space, yet are not required for validating new blocks. They are only retrieved when explicitly requested over the RPC API or when a connected peer attempts to bootstrap the chain. Pruning this data therefore yields substantial reductions in long-term storage requirements and allows execution clients to simplify their execution logic by removing code dedicated to processing legacy blocks and handling protocol upgrades. The EIP also changes how clients bootstrap to the chain. Instead of downloading the full historical record, a client can synchronize from a recent state using a valid weak-subjectivity

checkpoint, lowering initial bandwidth requirements and reducing the time needed for a new client to become fully operational [61]. At present, however, this proposal remains in a stagnant state and is not deployed on the Ethereum Mainnet, meaning that its potential benefits are not yet available to clients. The sparse client protocols presented in this work follow a complementary strategy. Rather than downloading all data and pruning it afterwards, sparse clients selectively retrieve, verify, and store only those transactions, state elements, or events that are relevant to the monitored accounts. Importantly, sparse clients *can* retain historical data if desired, without incurring prohibitive storage overhead of full nodes.

Clients for Lazy Ledgers Whereas EIP-4444 targets historical data retention within the existing protocol, other research explores alternative blockchain models that shift the division of responsibilities between validators and clients. Lazy blockchains decouple consensus transaction execution and validation from consensus. In this model, the primary responsibility of validators is limited to ordering transactions and ensuring their availability, rather than verifying their correctness. The burden of execution and validation is instead shifted to the clients that are directly interested in specific transactions, typically those associated with the applications they use [17, 58]. In this sense, sparse nodes and application-specific clients of lazy ledgers share certain similarities. However, application-specific clients in lazy ledgers must still download the entire ledger, sacrificing communication efficiency [3]. Sparse nodes achieve comparable properties while further reducing resource costs, since they selectively retrieve only the data relevant to the monitored accounts.

Rollup Clients In parallel, scalability efforts have increasingly turned to layer-2 systems, most prominently rollups. Rollups are L2 systems that improve blockchain scalability by posting their transaction data to a parent chain, while executing the corresponding state transitions off-chain. A rollup system typically consists of a parent chain, rollup full nodes, and rollup light clients. A sequencer—a specialized full node—collects transactions submitted by users, aggregates them, and posts the resulting batch to the parent chain. The parent chain ensures data availability, enabling any rollup full node to reconstruct the latest state independently of the sequencer—provided the rollup full node runs a light client of the parent chain. Rollup full nodes execute transactions in these batches, commonly referred to as rollup blocks. In contrast, rollup light clients do not download or execute the transaction data. Instead, they query full nodes for the most recent valid state [57]. This design achieves significantly higher throughput than the parent chain, but it also amplifies the verification burden for clients: the number of TPS is higher, and thus the fraction of non-relevant transactions grows even larger for most users. Both protocols presented in this work are fully compatible with EVM-based rollups. Instead of downloading and processing the full rollup ledger, a sparse node selectively re-executes only those transactions that touch the monitored substate.

Sharding While rollups build on top of an existing parent chain, blockchain sharding represents a complementary scaling technique at the consensus layer itself in which the network is partitioned into multiple parallel shards, each processing its own *subset* of transactions and maintaining local state. Unlike our work, sharding is a consensus-layer mechanism that scales block production by distributing the workload across multiple validator committees, each running a separate consensus for a single shard. This design reduces resource requirements for individual validators, as they only need to store and process a single shard instead of the entire blockchain [62]. In contrast, sparse clients operate on the client side and address the challenge of data consumption. A sparse client allows a user to efficiently monitor a specific subset of the blockchain without needing to track the state of an entire shard, or the full multi-shard network. Therefore, sharding and sparse clients are orthogonal: sharding partitions the network to help validators produce more data, while sparse clients enable users to verifiably access only the data they care about.

CHAPTER 9

Conclusion

This thesis presented Sparseth and Eventeth, two novel protocols that enable efficient and verifiable blockchain monitoring without requiring validator overhead. By operating exclusively at the execution layer and leveraging cryptographic commitments stored in smart contract state, our protocols provide a practical alternative to centralized infrastructure while maintaining strong security guarantees.

The formal analysis demonstrated that both protocols satisfy safety, liveness, and sparse validity under our adversarial model. Sparseth's interaction counter ensures no relevant transactions are omitted from the sparse ledger, while Eventeth's hash chain provides tamper-evident verification of event log completeness. These cryptographic primitives enable clients to autonomously verify the integrity and completeness of their local data without relying on trusted third parties.

Our implementation and evaluation quantified substantial efficiency gains: event nodes achieved 95% bandwidth reduction and nearly 99% storage reduction compared to full nodes, while sparse nodes achieved 92% computational reduction through selective transaction execution. Operational cost analysis revealed that sparse clients can reduce monthly expenses by 66-85% compared to full nodes. The gas overhead analysis showed modest costs of 4-16% for typical dApp transactions, which could be further mitigated through L2 deployment and optimized economic mechanisms.

Collectively, these results validate the practical feasibility of sparse node architectures for real-world deployment, demonstrating that secure and efficient blockchain monitoring is achievable.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Generative AI Tools Used

Use of ChatGPT In the preparation of this thesis, I made selective use of ChatGPT (OpenAI, GPT-5) as a supporting resource to assist with improving the presentation of the material I independently created. The primary purpose of using this tool was to refine the language of my writing, improve clarity, and ensure a high degree of linguistic consistency throughout the thesis. The conceptual foundations, research questions, theoretical framework, formal models, proofs, analyses, and technical contributions presented in this document were developed independently. ChatGPT was not used to generate ideas, derive arguments, construct proofs, or perform any form of original analysis—its role was strictly limited to improving the readability and cohesion of content that I had already written.

Furthermore, I maintained full control over the writing process and carefully reviewed, verified, and substantially revised all AI-assisted suggestions. In many cases, the outputs served only as a starting point for further refinement, and I made extensive modifications to ensure that the final formulations accurately represent my academic objectives, personal reasoning, and methodological rigor.

Use of GitHub Copilot For the implementation of this thesis, I made selective use of GitHub Copilot (GitHub, GPT-4.1), an AI-powered coding assistant. Copilot was primarily used to accelerate the software development process by providing code completions, syntax suggestions, and boilerplate implementations during the prototyping and testing of my project. While Copilot offered code snippets, I retained full responsibility for designing the architecture, defining the algorithms, and implementing the logic necessary to achieve the research objectives of the project.

I carefully reviewed, modified, and validated all generated code to ensure correctness, security, and consistency with the goals of my research. In many cases, the suggested code snippets served only as a starting point and underwent substantial adoption to fit the specific needs of the overall implementation. Consequently, the final implementation reflects my own engineering decisions, understanding, and problem-solving.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1	Illustration of an Ethereum Account	14
2.2	Illustration of a Blockchain	16
2.3	Illustration of the Ethereum Block Structure	17
2.4	Illustration of a State Transition	18
2.5	Illustration of an Ethereum Full Node	19
6.1	Illustration of the Sparse Node Architecture	68
7.1	DEX Market Share Ethereum Mainnet	80
7.2	Ethereum Mainnet Transaction Volume	81
7.3	Ethereum Sepolia Testnet Transaction Volume	83
7.4	Ethereum Sepolia DEX Benchmark Transaction Volume	84
7.5	Event Node Bandwidth Consumption	85
7.6	Sparse Node Bandwidth Consumption	86
7.7	Event and Sparse Node Storage Requirements	88
7.8	Sparse Node and Full Node Computation	89



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

5.1	Resource Consumption Comparison	59
7.1	Gas Cost Breakdown for Maintaining an Interaction Counter	75
7.2	Gas Overhead of the Interaction Counter Across Popular Ethereum dApps	77
7.3	Gas Cost Breakdown for Maintaining a Hash Chain Commitment	78
7.4	Gas Overhead of the Hash Chain Across Popular Ethereum dApps	79
7.5	Operational Costs of different Node Types	90



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

4.1	Eventeth Verifier Protocol	38
4.2	Eventeth Sequence Verification	40
4.3	Eventeth Prover Protocol	40
4.4	Sparseth Verifier Protocol	44
4.5	Sparseth Prover Protocol	46



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

- ABI** Application Binary Interface. 66
- API** Application Programming Interface. xi, xiii, 1, 18, 63, 67, 69, 71, 73, 84–87, 94
- AWS** Amazon Web Services. 73, 88, 90
- BAB** Byzantine Atomic Broadcast. 10, 11
- BAL** Block-Level Access List. 48
- BFT** Byzantine Fault-Tolerant. 10, 22, 23, 28, 34, 37
- CPU** Central Processing Unit. 89, 90
- DA** Data Availability. 4, 51, 52, 59–61
- DAO** Decentralized Autonomous Organization. 48
- dApp** Decentralized Application. xi, xiii, 2, 13, 34, 41, 48, 73–75, 77, 79–81, 97
- DeFi** Decentralized Finance. 76
- DEX** Decentralized Exchange. 35, 41, 76, 80, 87, 88
- EBS** Elastic Block Store. 90
- EC2** Elastic Compute Cloud. 90
- EIP** Ethereum Improvement Proposal. 18, 48, 94, 95
- EOA** Externally-Owned Account. 3, 14, 15, 18, 33, 42, 74
- ERC** Ethereum Request for Comments. 81, 82, 87
- ETH** Ether. 13
- EUF-CMA** Existential Unforgeability Under Chosen Message Attack. 22

EVM Ethereum Virtual Machine. xi, xiii, 2–4, 14, 18, 47, 49, 63, 64, 66, 79, 93, 95

FFG Friendly Finality Gadget. 15

Geth go-ethereum. 4, 63, 67, 89

GHOST Greedy Heaviest-Observed Sub-Tree. 15

GST Global Stabilization Time. 10, 24

IDE Integrated Development Environment. 74

IP Internet Protocol. 1

JSON JavaScript Object Notation. 18, 63, 67, 69, 70, 83, 85, 86

L1 Layer-1. 1, 2, 18, 49

L2 Layer-2. xi, xiii, 2, 48, 49, 79, 95, 97

LMD Latest Message Driven. 15

LTS Long Term Support. 82

MPT Merkle-Patricia Trie. 15–17, 70

NaaS Node-as-a-Service. xi, xiii, 1, 49

NFT Non-Fungible Token. 76

OFAC Office of Foreign Assets Control. 1

P2P Peer-to-Peer. 84, 94

PATRICIA Practical Algorithm To Retrieve Information Coded in Alphanumeric. 17

PKI Public Key Infrastructure. 23

PoC Proof of Concept. 4, 63, 67, 73, 85, 87

PoS Proof of Stake. 15, 18

PoW Proof of Work. 12

PPT Probabilistic Polynomial-Time. 23

RLP Recursive Length Prefix. 13

RPC Remote Procedure Call. 1, 2, 18, 48, 49, 63, 67, 69, 70, 83, 85, 86, 94

SMR State Machine Replication. 11, 12, 22

SPV Simple Payment Verification. 56

SSD Solid State Drive. 89

TEE Trusted Execution Environment. 49

TPS Transactions Per Second. xiii, 1, 95

USD United States Dollar. 90, 91

VM Virtual Machine. 82, 83



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

- [1] Vitalik Buterin et al. *EIP-4844: Shard Blob Transactions*. en. Feb. 2022. URL: <https://eips.ethereum.org/EIPS/eip-4844> (visited on 09/15/2025).
- [2] U.S. Department of the Treasury. *U.S. Treasury Sanctions Notorious Virtual Currency Mixer Tornado Cash*. en. Aug. 2022. URL: <https://home.treasury.gov/news/press-releases/jy0916> (visited on 09/18/2025).
- [3] Giulia Scaffino et al. *Sunfish: Reading Ledgers with Sparse Nodes*. Publication info: Preprint. 2024. URL: <https://eprint.iacr.org/2024/1680> (visited on 08/04/2025).
- [4] Vitalik Buterin. *A local-node-favoring delta to the scaling roadmap*. en. Mar. 2025. URL: <https://ethresear.ch/t/a-local-node-favoring-delta-to-the-scaling-roadmap/22368> (visited on 09/18/2025).
- [5] Yash Kamal Chaturvedi. *Expected EIPs in Glamsterdam Upgrade (Execution Layer)*. Aug. 2025. URL: <https://etherworld.co/2025/08/25/expected-eips-in-glamsterdam-upgrade-execution-layer/> (visited on 09/28/2025).
- [6] Toni Wahrstätter et al. *EIP-7928: Block-Level Access Lists*. en. Mar. 2025. URL: <https://eips.ethereum.org/EIPS/eip-7928> (visited on 09/28/2025).
- [7] A. J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. eng. 1st. OCLC: 44210973. Boca Raton: CRC Press, 2018. ISBN: 978-0-429-88131-2.
- [8] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. eng. Second edition. Chapman & Hall/CRC cryptography and network security. Boca Raton London New York: CRC Press, 2015. ISBN: 978-1-4665-7027-6.
- [9] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the Presence of Partial Synchrony”. en. In: *Journal of the ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/42282.42283. URL: <https://dl.acm.org/doi/10.1145/42282.42283> (visited on 08/05/2025).
- [10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. en. In: *Journal of the Association for Computing Machinery* 32.2 (Apr. 1985), pp. 374–283.

- [11] Nancy A. Lynch. *Distributed Algorithms*. The Morgan Kaufmann series in data management systems. San Francisco, Calif: Morgan Kaufmann, 1997. ISBN: 978-1-55860-348-6.
- [12] Srivatsan Sridhar et al. *Consensus Under Adversary Majority Done Right*. Publication info: Published elsewhere. Financial Cryptography and Data Security 2025. 2024. URL: <https://eprint.iacr.org/2024/1799> (visited on 08/09/2025).
- [13] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem”. en. In: *ACM Transactions on Programming Languages and Systems* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/357172.357176. URL: <https://dl.acm.org/doi/10.1145/357172.357176> (visited on 08/04/2025).
- [14] M. Pease, R. Shostak, and L. Lamport. “Reaching Agreement in the Presence of Faults”. en. In: *Journal of the ACM* 27.2 (Apr. 1980), pp. 228–234. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322186.322188. URL: <https://dl.acm.org/doi/10.1145/322186.322188> (visited on 08/10/2025).
- [15] Idit Keidar et al. “All You Need is DAG”. en. In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. Virtual Event Italy: ACM, July 2021, pp. 165–175. ISBN: 978-1-4503-8548-0. DOI: 10.1145/3465084.3467905. URL: <https://dl.acm.org/doi/10.1145/3465084.3467905> (visited on 08/10/2025).
- [16] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. eng. Cambridge: Cambridge University Press, 2008. ISBN: 978-0-521-87634-6 978-0-511-80531-8. DOI: 10.1017/CBO9780511805318.
- [17] Mustafa Al-Bassam. *LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts*. Version Number: 4. 2019. DOI: 10.48550/ARXIV.1905.09274. URL: <https://arxiv.org/abs/1905.09274> (visited on 08/11/2025).
- [18] Mohammad Mussadiq Jalalzai and Kushal Babel. *MonadBFT: Fast, Responsive, Fork-Resistant Streamlined Consensus*. Version Number: 1. Feb. 2025. DOI: 10.48550/ARXIV.2502.20692. URL: <https://arxiv.org/abs/2502.20692> (visited on 08/11/2025).
- [19] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. en. In: *ACM Computing Surveys* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/98163.98167. URL: <https://dl.acm.org/doi/10.1145/98163.98167> (visited on 08/11/2025).
- [20] Ittai Abraham et al. “Sync HotStuff: Simple and Practical Synchronous State Machine Replication”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 106–118. ISBN: 978-1-7281-3497-0. DOI: 10.1109/SP40000.2020.00044. URL: <https://ieeexplore.ieee.org/document/9152792/> (visited on 08/11/2025).

- [21] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. en. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Feb. 1999).
- [22] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”. en. In: (2008). URL: <https://bitcoin.org/bitcoin.pdf>.
- [23] Kaya Alpturer and S. Matthew Weinberg. *Optimal RANDAO Manipulation in Ethereum*. Version Number: 1. 2024. DOI: 10.48550/ARXIV.2409.19883. URL: <https://arxiv.org/abs/2409.19883> (visited on 08/12/2025).
- [24] Apostolos Tzinas, Srivatsan Sridhar, and Dionysis Zindros. “On-Chain Timestamps are Accurate”. en. In: *Financial Cryptography and Data Security*. Ed. by Jeremy Clark and Elaine Shi. Vol. 14744. Series Title: Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, Feb. 2025, pp. 110–127. ISBN: 978-3-031-78675-4 978-3-031-78676-1. DOI: 10.1007/978-3-031-78676-1_7. URL: https://link.springer.com/10.1007/978-3-031-78676-1_7 (visited on 08/13/2025).
- [25] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. “The Bitcoin Backbone Protocol with Chains of Variable Difficulty”. en. In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 291–323. ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_10. URL: http://link.springer.com/10.1007/978-3-319-63688-7_10 (visited on 08/30/2025).
- [26] Peter Todd. *BIP 65: OP_CHECKLOCKTIMEVERIFY*. en. Oct. 2014. URL: <https://bips.dev/65/> (visited on 08/30/2025).
- [27] Nick Szabo. “Formalizing and Securing Relationships on Public Networks”. en. In: *First Monday* (Sept. 1997). ISSN: 1396-0466. DOI: 10.5210/fm.v2i9.548. URL: <https://firstmonday.org/ojs/index.php/fm/article/view/548> (visited on 10/02/2025).
- [28] Vitalik Buterin. “Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.” en. In: (2014).
- [29] Gavin Wood. “Ethereum: A Secure Decentralised Generalised Transaction Ledger”. en. In: (Feb. 2025).
- [30] Mikhail Kalinin, Danny Ryan, and Vitalik Buterin. *EIP-3675: Upgrade Consensus to Proof-of-Stake*. en. July 2021. URL: <https://eips.ethereum.org/EIPS/eip-3675> (visited on 08/07/2025).
- [31] Vitalik Buterin and Virgil Griffith. *Casper the Friendly Finality Gadget*. arXiv:1710.09437 [cs]. Jan. 2019. DOI: 10.48550/arXiv.1710.09437. URL: <http://arxiv.org/abs/1710.09437> (visited on 08/10/2025).

- [32] Yonatan Sompolinsky and Aviv Zohar. *Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains*. Publication info: Preprint. MINOR revision. 2013. URL: <https://eprint.iacr.org/2013/881> (visited on 10/02/2025).
- [33] Vitalik Buterin et al. *Combining GHOST and Casper*. arXiv:2003.03052 [cs]. May 2020. DOI: 10.48550/arXiv.2003.03052. URL: <http://arxiv.org/abs/2003.03052> (visited on 10/02/2025).
- [34] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". en. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. Brisbane, QLD: IEEE, Apr. 2013, pp. 38–49. ISBN: 978-1-4673-4910-9 978-1-4673-4909-3 978-1-4673-4908-6. DOI: 10.1109/ICDE.2013.6544812. URL: <http://ieeexplore.ieee.org/document/6544812/> (visited on 10/02/2025).
- [35] Stuart Haber and W. Scott Stornetta. "Secure names for bit-strings". In: *Proceedings of the 4th ACM conference on Computer and communications security. CCS '97*. New York, NY, USA: Association for Computing Machinery, Apr. 1997, pp. 28–35. ISBN: 978-0-89791-912-8. DOI: 10.1145/266420.266430. URL: <https://dl.acm.org/doi/10.1145/266420.266430> (visited on 10/02/2025).
- [36] Micah Zoltu. *EIP-2718: Typed Transaction Envelope*. en. June 2020. URL: <https://eips.ethereum.org/EIPS/eip-2718> (visited on 08/13/2025).
- [37] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. *Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities*. Version Number: 5. 2018. DOI: 10.48550/ARXIV.1809.09044. URL: <https://arxiv.org/abs/1809.09044> (visited on 08/25/2025).
- [38] Shresth Agrawal et al. *Proofs of Proof-of-Stake with Sublinear Complexity*. Publication info: Published elsewhere. Minor revision. Advances in Financial Technologies - AFT 2023. 2022. URL: <https://eprint.iacr.org/2022/1642> (visited on 09/09/2025).
- [39] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. "Easy Impossibility Proofs for Distributed Consensus Problems". en. In: *Proceedings of the fourth annual ACM symposium on Principles of distributed computing - PODC '85*. Minaki, Ontario, Canada: ACM Press, 1985, pp. 59–70. ISBN: 978-0-89791-168-9. DOI: 10.1145/323596.323602. URL: <http://portal.acm.org/citation.cfm?doid=323596.323602> (visited on 08/11/2025).
- [40] Karl Wüst and Arthur Gervais. *Ethereum Eclipse Attacks*. en. Tech. rep. Artwork Size: 7 p. Medium: application/pdf. ETH Zurich, 2016. DOI: 10.3929/ETHZ-A-010724205. URL: <http://hdl.handle.net/20.500.11850/121310> (visited on 08/05/2025).
- [41] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. *Non-Interactive Proofs of Proof-of-Work*. Publication info: Preprint. MINOR revision. May 2018. URL: <https://eprint.iacr.org/2017/963> (visited on 08/05/2025).

- [42] Karl Wüst et al. *ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts*. Publication info: Published elsewhere. Minor revision. ACM CCS 2020. 2019. URL: <https://eprint.iacr.org/2019/835> (visited on 09/13/2025).
- [43] Leslie Lamport. “Password authentication with insecure communication”. en. In: *Communications of the ACM* 24.11 (Nov. 1981), pp. 770–772. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/358790.358797. URL: <https://dl.acm.org/doi/10.1145/358790.358797> (visited on 08/22/2025).
- [44] Pericle Perazzo and Riccardo Xefraj. “SmartFly: Fork-Free Super-Light Ethereum Classic Clients for Internet of Things”. In: *IEEE Internet of Things Journal* 11.9 (May 2024), pp. 15348–15358. ISSN: 2327-4662. DOI: 10.1109/JIOT.2024.3350333. URL: <https://ieeexplore.ieee.org/document/10388027/> (visited on 09/14/2025).
- [45] Mingchao Yu et al. *Coded Merkle Tree: Solving Data Availability Attacks in Blockchains*. Version Number: 2. Oct. 2019. DOI: 10.48550/ARXIV.1910.01247. URL: <https://arxiv.org/abs/1910.01247> (visited on 08/25/2025).
- [46] Mustafa Al-Bassam et al. “Fraud and Data Availability Proofs: Detecting Invalid Blocks in Light Clients”. en. In: *Financial Cryptography and Data Security*. Ed. by Nikita Borisov and Claudia Diaz. Vol. 12675. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 279–298. ISBN: 978-3-662-64330-3 978-3-662-64331-0. DOI: 10.1007/978-3-662-64331-0_15. URL: https://link.springer.com/10.1007/978-3-662-64331-0_15 (visited on 08/25/2025).
- [47] Mathias Hall-Andersen, Mark Simkin, and Benedikt Wagner. *Foundations of Data Availability Sampling*. Publication info: Published by the IACR in CIC 2024. 2023. URL: <https://eprint.iacr.org/2023/1079> (visited on 08/25/2025).
- [48] Andreas M. Antonopoulos and Gavin A. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. eng. First edition. EBL-Schweitzer. Beijing, Boston, Farnham, Sebastopol, Tokyo: O’Reilly, 2019. ISBN: 978-1-4919-7194-9 978-1-4919-7191-8.
- [49] Massimo Bartoletti et al. *Smart Contract Languages: A Comparative Analysis*. Version Number: 2. 2024. DOI: 10.48550/ARXIV.2404.04129. URL: <https://arxiv.org/abs/2404.04129> (visited on 08/31/2025).
- [50] Maximilian Wohrer and Uwe Zdun. “Smart contracts: security patterns in the ethereum ecosystem and solidity”. en. In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. Campobasso: IEEE, Mar. 2018, pp. 2–8. ISBN: 978-1-5386-5986-1. DOI: 10.1109/IWBOSE.2018.8327565. URL: <http://ieeexplore.ieee.org/document/8327565/> (visited on 08/31/2025).
- [51] *Solidity Programming Language*. en. URL: <https://soliditylang.org/> (visited on 08/31/2025).

- [52] Vitalik Buterin and Martin Swende. *EIP-2929: Gas Cost Increases for State Access Opcodes*. en. Sept. 2020. URL: <https://eips.ethereum.org/EIPS/eip-2929> (visited on 08/26/2025).
- [53] *Etherscan - The Ethereum Blockchain Explorer*. en. URL: <https://etherscan.io/> (visited on 08/26/2025).
- [54] Fabian Vogelsteller and Vitalik Buterin. *ERC-20: Token Standard*. en. Nov. 2015. URL: <https://eips.ethereum.org/EIPS/eip-20> (visited on 08/29/2025).
- [55] William Entriken et al. *ERC-721: Non-Fungible Token Standard*. en. Jan. 2018. URL: <https://eips.ethereum.org/EIPS/eip-721> (visited on 08/29/2025).
- [56] Noah Zinsmeister, Dan Robinson, and Adams Hayden. “Uniswap V2 Core”. en. In: (Mar. 2020).
- [57] Ertem Nusret Tas et al. *Accountable Safety for Rollups*. arXiv:2210.15017 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2210.15017. URL: <http://arxiv.org/abs/2210.15017> (visited on 09/20/2025).
- [58] Ertem Nusret Tas et al. *Light Clients for Lazy Blockchains*. arXiv:2203.15968 [cs]. May 2024. DOI: 10.48550/arXiv.2203.15968. URL: <http://arxiv.org/abs/2203.15968> (visited on 08/04/2025).
- [59] Psi Vesely et al. *Plumo: An Ultralight Blockchain Client*. Publication info: Published elsewhere. Minor revision. *Financial Cryptography and Data Security 2022*. 2021. URL: <https://eprint.iacr.org/2021/1361> (visited on 09/29/2025).
- [60] Lukas Aumayr et al. “Blink: An Optimal Proof of Proof-of-Work”. en. In: (2024).
- [61] George Kadianakis, lightclient, and Alex Stokes. *EIP-4444: Bound Historical Data in Execution Clients*. en. Nov. 2021. URL: <https://eips.ethereum.org/EIPS/eip-4444> (visited on 09/16/2025).
- [62] Georgia Avarikioti et al. *Divide and Scale: Formalization and Roadmap to Robust Sharding*. arXiv:1910.10434 [cs]. May 2023. DOI: 10.48550/arXiv.1910.10434. URL: <http://arxiv.org/abs/1910.10434> (visited on 09/29/2025).