Harmonia: Securing Cross-Chain Applications Using Zero-Knowledge Proofs

Rafael Belchior, Dimo Dimov, Zahary Karadjov,
Jonas Pfannschmidt, André Vasconcelos, Miguel Correia

Instituto Superior Técnico INESC-ID Blockdaemon Metacraft Labs

CONTENTS

***Abstract*—The field of blockchain interoperability plays a pivotal role in blockchain adoption. Despite these advances, a notorious problem persists: the high number and success rate of attacks on blockchain bridges.**

We propose Harmonia, a framework for building robust, secure, efficient, and decentralized cross-chain applications. A main component of Harmonia is DendrETH, a decentralized and efficient zero-knowledge proof-based light client. DendrETH mitigates security problems by lowering the attack surface by relying on the properties of zero-knowledge proofs. The DendrETH instance of this paper is an improvement of Ethereum's light client sync protocol that fixes critical security flaws. This light client protocol is implemented as a smart contract, allowing blockchains to read the state of the source blockchain in a trust-minimized way. Harmonia and DendrETH support several cross-chain use cases, such as secure cross-blockchain bridges (asset transfers) and smart contract migrations (data transfers), without a trusted operator. We implemented Harmonia in 9K lines of code.

Our implementation is compatible with the Ethereum Virtual Machine (EVM) based chains and some non-EVM chains. Our experimental evaluation shows that Harmonia can generate light client updates with reasonable latency, costs (a dozen to a few thousand US dollars per year), and minimal storage requirements (around 4.5 MB per year). We also carried out experiments to evaluate the security of DendrETH. We provide an open-source implementation and reproducible environment for researchers and practitioners to replicate our results.

I. INTRODUCTION

With the development of increasingly more complex cross-chain logic, the trend is for the major *decentralized applications* (dApps) to go either *cross-chain* or *multi-chain* [1], where various chains coexist, sharing data and digital assets. This allows developers and users to choose the best infrastructure based on trade-offs (e.g., cost, performance, and convenience [2], [3]). This enables workflows supported by different infrastructure components that process *data transfers*, *asset transfers*, and *asset exchanges*, the so-called *interoperability modes* [2]. In practice, realizing these interoperability modes is orchestrating a set of coordinated reads and writes of transactions settled in different blockchains. Data transfers are implemented by arbitrary message-passing bridges [4], oracles [5], [6], and blockchain gateways [7]–[9]; cross-chain bridges typically implement asset transfers [10], [11]; asset exchanges are implemented by hash time lock contracts or liquidity networks [1]. Asset transfers are particularly vulnerable. In fact, current bridge implementations are insecure to the point of having caused around $3B in losses [12]–[14]. The causes include large attack surface, lack of transparency, poor monitoring techniques, lack of incident response plans, reliance on a single point of failure (either individual nodes or committees) [15], cybersecurity attacks that steal private keys [16], bad operational practices [17], attacks on economic incentives [18], and others [10], [12], [14].

A. Problem and Solution Overviews

Academia and industry agree that *interoperability mechanisms* (IMs) relying native verification of transactions across the *source* and *destination (or target) blockchains* are the safest [1], [19], [20]. Bridges allow the transfer of funds by providing facts on the source chain and relaying those to the destination chain. The destination chain independently verifies that the received state is valid and final according to the state transition and consensus rules of the source network. For example, a user can *lock* (or *burn*) some tokens in a source chain, and the bridge can *mint* the corresponding amount of that asset on the destination chain provably (asset transfer). To prove a transaction is valid on an external blockchain, cross-chain applications use light clients [14].

A *light client protocol* (or simply light client) is a protocol that allows proving the inclusion of a transaction in a blockchain without downloading the full blockchain (typically only the block headers [21], or a subset of them [22]). This requires, e.g., in the case of Ethereum, that 1) there is a valid block header that includes the transaction to be proved and 2) a valid *Merkle proof* against the block header root is provided. The sequence of transactions happening in a blockchain creates a ”history”, that is appended to the blocks that form the blockchain. However, light clients allow multiple valid histories to be validated as long as they respect consensus rules. This creates an attack vector that, combined with the lack of incentives for different parties to behave correctly, can lead to exploiting the light-client system. We call this exploitation *Ghost Checkpoint Attestation Attack*. We explain this attack and a possible solution later in the paper.

In this work, we propose *Harmonia*[1], a framework to build reliable cross-chain applications, realizing data or asset transfers. At its core, Harmonia leverages a light client protocol, assuring the safety of interoperability. Our implementation allows us to prove facts on the direction *Ethereum* $\rightarrow$ *other blockchains*. An additional contribution is *DendrETH*, an improved version of Ethereum's light client protocol [23] that prevents the *Ghost Checkpoint Attestation Attack*. Introduced in the Altair hard fork, the Altair Light Client protocol suffered from critical security vulnerabilities. DendrETH takes as input a *zero-knowledge proof*, specifically, a succinct non-interactive argument of knowledge, or SNARK [24]–[26], which allows verifying on-chain the light client protocol rules in a cost-efficient way.

The key intuition of the paper is that SNARKs can be used to prove that DendrETH rules are correctly executed, which is useful because SNARKs can be verified on-chain. If the verification is successful, the light client is updated. The output of the light client update is a validated *block header*. In this way, cross-chain applications can verify the state of other chains by using different cross-chain proof mechanisms [27] such as Merkle proofs [28] against the updated block header. This allows cross-chain applications to have guarantees on the cross-chain state, e.g., transaction inclusion, and to perform arbitrary *cross-chain logic* (rules that orchestrate cross-chain transactions[2]) [12]. Therefore, we lay the infrastructure to build trustless cross-chain applications (dApps), without the need to depend on external parties to verify transactions, thus eliminating a single point of failure and improving interoperability security.

Figure 1 shows the high-level architecture of our framework. The starting point is two cross-chain smart contracts, A and E. Contract E will have `read` and/or `write` dependencies on A, depending on the defined cross-chain logic. In steps 0 and 1, user B interacts with the source chain smart contract A directly or through the *Application Relayer* C (respectively), issuing transactions that change the local state of A; therefore, we deem the relayers blockchain clients. The Application Relayer (C) will see changes to the contract in step 2 and create a Merkle proof that attests to the state changes that step 1 triggered. The Merkle proof and use-case-specific data are sent to E according to the cross-chain logic.

In parallel, the *SNARK Relayer* D gathers the state and data of the light client (step 3) to create a SNARK proof that proves a light client update. The SNARK, along with the necessary input data are sent to the light client verifier contract G, which is a SNARK verifier contract (step 4). If the verification succeeds, the necessary data to validate Merkle proofs will be available on the smart contract to be consumed by applications. The Application Relayer can transact with E following a determined cross-chain logic (step 5). The cross-chain logic contract on the target chain will only execute the logic if the Merkle proof verification on the Application Proof Verifier Contract (contract F) succeeds (step 6). Contract F calls the SNARK verifier to obtain validated state roots to run the Merkle proofs against.

B. Problem Definition

Let us consider a pair of chains A and B, a light client protocol $\mathcal{L}$, and cross-chain rules $\mathcal{R}$. The problem at hand is decomposed into two sub-problems.

The first problem is to prove the validity of a block header from blockchain A on blockchain B succinctly and cheaply (without having access to the whole blockchain, which would be impractical [29]). Proving the validity of a block header further proves the validity of any transactions included in that block header. Conveniently, SNARKs provide the necessary properties for this. Solving this problem provides us with a validated block header, which contains valid state roots. We will show that this problem is aggravated by the fact that, currently, the group of nodes running $\mathcal{L}$ in the case of Ethereum can create and propagate multiple conflicting variations of the same history, conducting a new type of attack with repercussions on the cross-chain ecosystem connected to Ethereum.

The second problem is to prove a claim on chain A (in terms of reads and writes), such that chain B can be convinced and allow the execution of arbitrary logic $\mathcal{R}$. Such proofs are, for example, Merkle proofs and are rooted in the validated state root. Solving the proposed problems implies solving a set of technical challenges.

C. Technical Challenges

We aim to provide a light client protocol that addresses the following key challenges ($\mathcal{C}$), some on-chain and the last off-chain:

- On-chain **Safety Failure** ($\mathcal{C}_1$): A light client that has been compromised presents risks to cross-chain use cases. A compromised light client can lead to the validation of invalid block headers, which can then be submitted and executed on a remote deployment. We provide a solution for this challenge in Section VI-F.
- On-chain **Safety Failure Propagation** ($\mathcal{C}_2$): If the light client is compromised, an invalid message could be created and executed on a remote deployment with immediate effect, giving no time for stakeholders to react. We handle this challenge in Section H.
- On-chain **Accountability** ($\mathcal{C}_3$): The source chain light client should only be able to sign one light client update (one valid version of the history) at any time. Relayers should detect on-chain misbehavior in the form of multiple signatures, so that it is punished by slashing an amount of collateral. We solve this challenge in Section III-F.
- Off-chain **Liveness Failure** ($\mathcal{C}_4$): Failure of the source chain for long periods could significantly delay or altogether

[1] Named after the Greek god of harmony and concord, Harmonia provides harmonious and reliable integration of different blockchains.

[2] Example rule: "after a lock event on the source blockchain, an unlock event should happen in the target blockchain, within a certain time interval."

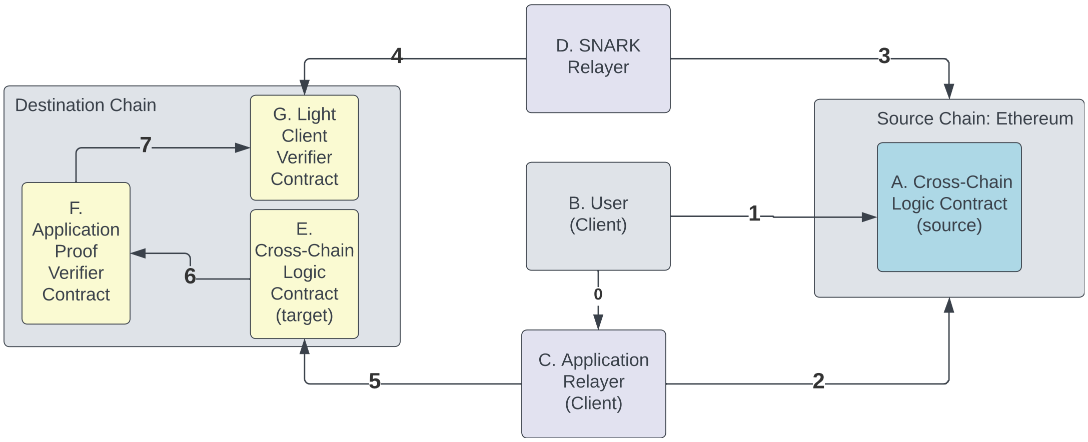

D. SNARK Relayer
4
3
Destination Chain
G. Light Client Verifier Contract
7
F. Application Proof Verifier Contract
E. Cross-Chain Logic Contract (target)
6
Source Chain: Ethereum
A. Cross-Chain Logic Contract (source)
B. User (Client)
1
0
C. Application Relayer (Client)
5
2

Fig. 1: Harmonia's high-level architecture. The cross-chain contract is represented by 🟦. Relayer components 🟪. Destination chain components 🟨.

prevent updates on the target chain. These liveness failures come in the form of occasional intermittent interruptions that lead to delayed or overlooked messages. We provide a solution for this challenge in Section VI-F.

The existing technical challenges hinder the development of solutions that can create and verify SNARKs on-demand [30] across a wide range of use cases.

D. Contributions

This paper provides a safer way to implement cross-chain applications by minimizing the attack surface, learning from the past while focusing on high performance and cost reduction. In particular:

- We propose Harmonia, a framework to build robust, secure, efficient, and decentralized cross-chain applications based on SNARK-based light clients.
- We formalize the properties of Altair Light Client (ALC), the canonical light client protocol for the Ethereum 2.0 network, as well as specify, formalize, and present an improvement that strengthens the cryptoeconomic security of the ALC, which we call DendrETH. To our knowledge, we are the first to propose improvements that fix critical issues in ALC security.
- We implement and experimentally evaluate Harmonia instantiated with DendrETH in terms of latency, throughput, and costs. We implement a data transfer use case using Harmonia and DendrETH. After that, we provide a qualitative evaluation that addresses the problems and technical challenges of this new technology: security, availability, censorship resistance, and others. This includes simulations of long-range attacks that may affect ALC and DendrETH.

E. Outline

This paper is organized as follows: in Section II we introduce the background on blockchain, cryptography, SNARKS, and cross-chain applications. Section III presents Harmonia, namely the system model and actors, threat and network model, system goals, ALC, and DendrETH. After that, we present our framework for building cross-chain applications using SNARK-based light clients. Next, Section IV showcases the implementation details of the relayers and verifier smart contracts. In Section V, we present the empirical evaluation. Section VI presents the qualitative evaluation, discussion, security proofs, and future work. Section VII presents the related work. Section VIII concludes the paper. A set of appendices complements the paper.

II. PRELIMINARIES

This section introduces the background. We build on the notation of previous work [27]. Let λ_s be the security parameter and λ_l be the liveness parameter. Let $x[i]$ denote the i^{th} element in a vector x.

A. Blockchain

We consider a ledger (or blockchain) $\mathcal{L}$ a versioned key-value store. The block number (also called the *height* of the blockchain) allow us to version the key-value store. Reading key a at version/block number b from ledger $\mathcal{L}$ is represented by $read_{\mathcal{L}}(a)[b]$, and writing value v at a by $write_{\mathcal{L}}(a, v)$. If b is omitted, we are reading the latest state (i.e., maximum height). Blockchains store their data in blocks, and we follow the definition from [2].

Definition 1. *Secure ledger. A ledger is deemed secure if it satisfies three properties:*

- *Consistency*: honest nodes possess a large common prefix, i.e., if n honest parties prune x blocks from their local chains, the probability that the resulting pruned chains will not be mutual prefixes of each other drops exponentially with the number of blocks belonging to the common prefix.
- *Chain quality*: there is an upper bound on the ratio of blocks proposed to the chain of any honest party n contributed by malicious parties.

- *Liveness*: if an honest node receives a valid transaction, it is eventually included in the blockchain by all honest nodes.

For these properties to hold, it is required that the number of malicious nodes f is bounded by the number of honest nodes n - the typical byzantine fault tolerance threshold is $n > 3f + 1$. Malicious parties, also called Byzantine, may act arbitrarily, while honest parties follow the protocol. These properties apply to the generality of ledgers - in a private ledger, however, chain quality is typically 100% as parties are not malicious or, if they are, they are held accountable [31], and ensure correct state transitions.

Definition 2. *State. The state of a blockchain is a tuple* $(s_k, s_{k,v}, \mathcal{T}, \pi_k)$*, where* s_k *is the state key,* $s_{k,v}$ *the value corresponding to that key, at the latest block number,* $\mathcal{T}$ *a list of transactions and* π_k *a list of proofs. We denote the global state by the union of all state tuples at time* t*):*

$$\mathcal{S}_t = \bigcup_k s_{k,v}$$

We can derive the state of a blockchain at different times by computing all the transactions since the initial state. Transactions trigger state changes in smart contracts. For users to interact with smart contracts, they need to sign transactions such that their payload targets a specific smart contract. Users pay a tax on executing programs on the decentralized network of nodes called *gas*. There is a dynamic market rate between gas and Ether (ETH), the native cryptocurrency used in the Ethereum network that depends on supply and demand. Gas is used as an incentive for users to do moderate their use of blockchain resources (since all validator nodes do the same computation, and database/storage is replicated across all full nodes). Some operations are more expensive than others. For instance, an Ether transfer in Ethereum costs 21K gas, while verifying an EdDSA signature costs about 500K gas [32], about 1.6 USD and 40 USD, respectively, at the time of writing (June 2023). Storing 1MB of data costs about 655 million gas, around 50K USD [33]. However, reading data is free. While adding a natural barrier to transaction scalability, imposing a gas limit helps reduce the amount of irrelevant information on the blockchain and prevents Sybil attacks [34]. In this paper, we will study the application of specific types of proofs to offload computation off-chain, to reduce on-chain computation and storage.

The source chain (or source blockchain) is the chain about which we want to prove facts and execute custom cross-chain logic (on a target chain or destination chain) [4].

B. Cryptographic Building Blocks

1) Cryptographic Keys: Accounts of a blockchain are tuples (K_k^{id}, K_P^{id}, id), capable of reading and writing to a ledger via a blockchain client (or node), where [35]:

- K_k^{id} is a private key, used as the signing key.
- K_P^{id} is a public key, used as the verifying key.
- id is the unique identifier of the participant. It is the output of a function over the participant's public key.

2) Hash Functions: A cryptographic hash function is a function that takes an input and returns a fixed-size string of bytes, typically called a hash value or digest. The output is unique, with overwhelming probability to each unique input (one-way function). Given a hash h, it is computationally infeasible to find an input value x such that $hash(x) = h$ (pre-image resistance property) or to find two different inputs that hash to the same output (collision resistance). These properties make hash functions essential in various cryptographic operations, such as digital signatures and message integrity checks [36].

3) Signatures: A signature is a mathematical scheme for verifying the authenticity of messages. A node N can create a signature σ via algorithm $\texttt{SIGN}$ over a message m using its private key, i.e., $\texttt{SIGN}_{K_k^n}(m) \rightarrow \sigma$. To verify a signature σ, a verifier can run $\texttt{VERIFY}$, taking as input a signature, a message, and the public key of the signer, i.e., $\texttt{VERIFY}_{K_p^n}(m, \sigma) \rightarrow \{0, 1\}$.

Aggregate signatures [37] are digital signatures where a set of nodes uses a mathematical function to combine their signatures $\{\sigma_1, \sigma_2, \ldots, \sigma_N\}$ into a single signature σ_{1-N}. A widely used aggregate signature scheme is BLS (Boneh-Lynn-Shacham) [38], which is also used in Ethereum [39]. Verification of a BLS signature is computationally expensive compared with verification of an ECDSA signature. The advantage of aggregate signatures in the context of this paper is that aggregate signatures provide a succinct way to represent a set of signatures in constant size. Signatures can also be aggregated with SNARKs (which we will discuss later).

4) Accumulators: Accumulators enable succinct (space and time efficient) and binding representation of a set of elements, supporting proofs of membership (or non-membership) [40], [41]. Accumulator operations include adding an element, creating a membership proof π that element v is in the accumulator set s, and verifying π. Removing elements and proofs of non-membership are optional and depend on the specific implementation. Next, we present Merkle trees, a type of accumulator.

C. Merkle Trees and Merkle Proofs

Merkle trees are accumulators implemented as binary trees. The leaves of the tree are data items, and the parent nodes are hashes of that data. The Merkle tree is calculated recursively: each parent element is the hash of its two children until the root is obtained. The tree's root is a succinct vector commitment to a state at a certain time. Merkle trees allow proving data inclusion. In particular, to construct a proof, we include all the nodes along the path needed to allow a recursive hashing up to the tree root (i.e., hashes of sibling nodes that are not in the direct path).

Definition 3. *Merkle proof. A Merkle proof (proof of inclusion) is a path between the tree's root and a leaf node. Given a vector* v *of* i *elements, we have three algorithms [32]:*

- $root \leftarrow tree.\texttt{commit}(v)$.
- $(v[i], \pi_i) \leftarrow tree.\texttt{proof}(v, i)$.

- $\{0,1\} \leftarrow tree.\texttt{verify}(\pi_i, root, v[i])$.

The `commit` algorithm adds an element to the Merkle tree. The `proof` algorithm creates a Merkle proof (a path) for the i^{th} element of v. The `verify` algorithm verifies the proof. Appendix B shows the Merkle proof verification algorithm.

D. Light Client Protocol

Ethereum offers a way to verify whether a transaction is included in a block using only the block header, avoiding downloading the full block. This mechanism is called simplified payment verification, [4], [22], [42], which evolved into the concept of light client. A light client is a lightweight version of a full blockchain node that allows participants to interact with the network without downloading the entire blockchain. Light clients can perform queries on the blockchain, e.g., queries on the balance of an account [43]. Light clients validate the consensus on the inclusion of transactions on the chain, but not the validity of the transactions themselves (see Appendix C-A). This process is done by verifying that a block has a supermajority of attestations from validators. These light clients will download light client updates (which we define formally soon) from Ethereum full nodes using the Beacon REST API that nodes provide. That way, transaction inclusion can be proven, e.g., a user can be convinced that the transaction transferring funds to a certain account was indeed included in the blockchain.

Formally, we denote the i^{th} block header of a blockchain by $\texttt{BlockHeader}_i$ and a chain of block headers from i to j by $\texttt{HeaderChain}_i^j$. We denote the last available, finalized block header by $\texttt{BlockHeader}_{|HeaderChain|}$. The light client state at sync period i is denoted $\mathcal{L}_{\mathcal{S}_i}$ and contains the current block header at the beginning of period i, the current sync committee, the next sync committee, and data.

Definition 4. *Light client. A light client $\mathcal{L}$ is an algorithm that has the following primitives [43]:*

- $\texttt{INIT}(\texttt{BlockHeader}_i) \rightarrow (\mathcal{L}_{\mathcal{S}_i}, \pi)$: The light client takes as input a bootstrap block (either genesis block or a pre-agreed block), and initializes the state with an interactive protocol with a full node and receives a proof π of correct initialization. We assume the bootstrap block is trusted (either by social consensus or cryptographically).
- $\texttt{QUERY}(\mathcal{L}_{\mathcal{S}_i}, data) \rightarrow (resp, \pi)$: a client can read the global state of the light client (and, indirectly, the global state of the underlying blockchain). The light client returns a response $resp$ and a proof π that authenticates the response or an error $\perp$.
- $\texttt{LCU}(\mathcal{L}_{\mathcal{S}_r}, data) \rightarrow (\mathcal{L}_{\mathcal{S}_{r+1}}, \perp)$: a light client update `LCU` takes as input the current light client state $\mathcal{S}$ and auxiliary data, and outputs the next light client state or an error.

Note that $resp$ and $data$ vary according to the specific light client protocol. In some cases, $data = \{\texttt{BlockHeader}_{r+1}, ...\}$.

Definition 5. *Secure and Efficient Light Client. A secure and efficient light client is a client that respects the following properties:*

- Soundness: After `INIT`, a malicious adversary should not be able to convince a light client $\mathcal{L}$ to accept a forged transaction. On the other hand, the adversary should not be able to convince $\mathcal{L}$ not to accept a valid transaction.
- Liveness: Valid transactions received by an honest full node are eventually included in the chain. A light client protocol eventually includes such transactions in a block header. This means that `QUERY` returns up-to-date requests up to a liveness parameter λ_l.
- Succinctness: For each state update, the light client protocol takes linear time to synchronize the state. `INIT` and `LCU` computation and communication are sublinear to the size of the blocks.

E. Altair Hard Fork and the Ethereum Sync Committees

Altair [39] is the first hard fork of Ethereum. This update brings two major changes to the previous mode of operating: 1) how rewards and penalties are calculated for validators, and 2) support for light clients, which will in turn add an additional reward type. This second change introduces a *sync committee* as the base layer to implement a light client protocol called *Altair Light Client* (ALC). The committee consists of 512 validators, randomly selected every sync committee period lasting 256 epochs (≈ 27 hours). Nodes are given 512 epochs of prior notice before they become sync committee members. A list of the committee members is saved in the state of the Beacon chain. The sync committee signs new block headers, so Ethereum light clients can verify Ethereum's state using a Merkle proof. This process authenticates more recent block signatures using the sync committee's public keys. For more details on the Altair Hard Fork and the sync committee, please consult Appendix C

F. SNARKs

Broadly speaking, Succinct Non-Interactive Argument of Knowledge (SNARKs) are proofs that a statement is true [25], [44]. In recent years, their popularity increased as several key applications for blockchain were recognized, namely, for increasing scalability, interoperability, and privacy (zero-knowledge proofs [45], [46]). We explore the application of this technology in the context of blockchain interoperability. By proving the validity of a block header via a SNARK, cross-chain logic on a target blockchain (in the form of a decentralized application/smart contract) can now prove the state from a source blockchain using Merkle proofs that are verified against the state root of the validated beacon block header.

A SNARK is a succinct claim on a predicate. Succinct means that the size of the proof is smaller than the computational process it represents (typically a few kilobytes); likewise, the time to verify the proof is sublinear to the size of the proof (typically a few milliseconds). Non-interactive means no back-and-forth interactions between the prover (Harmonia

↔ relayers, in our case) and the verifier (verifier smart contract that is part of the light client). This way, the prover can convince the verifier with a single message that a valid computation (coherent input-output relation) occurred. This allows offloading costly computation off-chain while preserving the integrity necessary for a decentralized system.

SNARKs can represent computations over arithmetic circuits, a model for computing polynomials. Arithmetic circuits, in turn, are represented as directed acyclic graphs where each node is an arithmetic operation, and the edges are inputs to that operation. The circuit consists of addition gates, multiplication gates, and some constant gates. In the same way, boolean circuits carry bits in wires, arithmetic circuits carry integers. The idea is then to represent programs as arithmetic circuits and to generate SNARKs proving computation over those circuits. Another way to represent circuits is via an R1CS (Rank-1 Constraint System), which defines linear constraints over the vectors representing the program. While explaining the technical nuances of SNARKs is beyond the scope of this paper, we provide further details on SNARKs, including a formalization, trusted setup, generation, verification, and proof systems, in Appendix D.

Different tools and frameworks exist to create SNARKs. Those tools allow the definition of programs (typically written in a domain-specific language) that are compiled into intermediary representations (e.g., arithmetic circuits, R1CSs). This is typically called the *frontend* of a SNARK system. The frontend is responsible for circuit definition and witness generation. The prover generates a proof with the witness and sends it to the verifier (see Appendix D). These two algorithms and the setup constitute the *backend* of a SNARK proving system. The backend uses a particular proof system. As an example, that we use in this paper, Groth16 [47], [48] is widely recognized for its efficiency, making it one of the most popular SNARK schemes used in blockchain.

G. Cross-chain Transactions / Logic / State

We derive the definitions of cross-chain transactions and cross-chain state from recent research [1], [12]. A cross-chain transaction is a set of local transactions (e.g., one transaction in Ethereum and one transaction in Polkadot), governed by cross-chain rules. Cross-chain rules (or cross-chain logic, denoted by ζ) are the dependencies between local transactions. For example, a transaction with payload $p = (lock, amount, destinationaddress, proof)$ in Ethereum should have a transaction in Polkadot with payload $p' = (mint, amount, destinationaddress, proof)$. Cross-chain rules may be enforced by off-chain relayers and smart contracts. We call the state that a set of cross-chain transactions generate the *cross-chain state*. A cross-chain state is a key-value store that spawns across the blockchains that process cross-chain transactions. This state stores information useful to execute cross-chain logic. In our bridge example, the state would be a ledger storing $(p, p', ...)$ for bookkeeping. This work showcases the relevance of these concepts for our use case and how our system plays a role in realizing them.

III. THE HARMONIA FRAMEWORK

In this section, we explain how the Harmonia framework works and how one can implement interoperable dApps on top of it. Harmonia is a framework to build reliable cross-chain applications, realizing the three interoperability modes [2]. At its core, it uses a light client protocol for the safety of interoperability. Our instantiation of Harmonia uses DendrETH as an on-chain SNARK-based light client that allows proving facts on the direction Ethereum → other chains, although Harmonia can support other combinations.

A. System Model and Components

Harmonia establishes a unidirectional communication channel between blockchains[3], which typically have different consensus mechanisms and trust models. A basilar assumption is that the *underlying blockchains are secure ledgers* (*cf.* Section II-A), otherwise, it is not possible to provide guarantees about the security of a mechanism that provides interoperability between them.

A transaction has achieved finality if, for a block at index i, the difference between the head of the chain, block j, and block i is higher than a liveness parameter λ_l, i.e., $j - i > \lambda_l$. The liveness parameter of the source chain is particularly relevant to the destination blockchain because cross-chain logic depends on transactions from the source chain. In practice, the destination chain needs to wait at least λ_l[4].

Harmonia includes several agents. The simplified architecture is in Figure 1, and a more detailed version in Figure 2:

- **Source and Destination chains** (cf. Section C-1): we assume chains support smart contracts.
- **Light Client Verifier Contract**: a smart contract deployed on the destination chain that verifies SNARKS created off-chain. It exposes a list of validated execution roots, optimistic roots, and finalized header roots that cross-chain applications can consume.
- **Application Verifier Contract**: a smart contract deployed on the destination chain that takes as input Merkle proofs and verifies them against the latest validated block root made available by the Light Client Verifier Contract.
- **Cross-chain Logic Contracts**: a pair of smart contracts. The first one is deployed on the source chain and holds business logic belonging to that chain. The second contract is deployed on the second chain and enforces cross-chain rules based on the state of the first contract, having a dependency on the Application Light Client Verifier Contract for validating updates.
- **SNARK and Application Relayers**: relayers read the global state and require a full node (or blockchain client access). For a finalized ledger, every client will read the same state. Two types of relayers are considered.

---

[3] one can spawn two Harmonia unidirectional communication channels in opposite directions, creating a de-facto bi-directional channel.

[4] Otherwise, proven facts on the source chain could be reverted, jeopardizing the cross-chain logic [12], [49].

The SNARK relayer creates SNARKS from on-chain information on the source chain and relays it to the verifier contract on the destination chain. The Application Relayer creates collects relevant data to perform interoperation, accompanied by a proof (e.g., Merkle proof) - and submits it to the cross-chain logic contracts.

Agents send arbitrary, authenticated messages between one another, e.g., transaction payloads, ACKS, and encoded API calls. The system considers two *actors*: the end user who interacts with a frontend connected to an interoperable system and an adversary that tries to steal user funds by attacking the interoperable solution.

B. Threat and Network Model

The adversary $\mathcal{A}$ is a probabilistic polynomial time algorithm that can perform various actions, namely corrupting a subset of the validators before the protocol execution commences. Adversarial validators on the source blockchain are denoted by f and the total number of nodes by n (honest nodes are assumed to be $n - f$ and typically $n > 3f + 1$, depending on the specific security threshold necessary for a chain to be considered secure). We assume the cryptographic primitives of the source and target blockchains are secure (hash functions, signing algorithms, communication channels, public-key infrastructure). We assume the source and target blockchains are secure (cf. Section C-1). Adversaries are computationally bounded. We rely on several cryptographic assumptions, namely the hardness of the discrete logarithm problem [50], the random oracle model [51], the common reference string model [52], and the non-falsifiable assumptions [53]. Under these constraints, sound SNARKs exist for any NP statement [52]–[54] - including for the high-level statement “the rules of the light client sync protocol update are correctly applied.”

Honest nodes from both chains have reliable and secure communication channels. The adversary controls the message delivery schedule and observes messages before the intended recipients. If a client is offline, upon its recovery, it observes all the messages it missed while offline. The network model considered is partially synchronous (for every message sent, an adversary can arbitrarily delay it up to a certain threshold). This implies no long network partitions, allowing the light client to have live updates. The threat model of Harmonia is tied to the security and crypto-economic models of the light client protocol and, if applicable, its light sync committee. Finally, we note that the security of our model is tied to the security of the proof system we use, which may rely on one or more of the referred cryptographic assumptions.

C. System Goals

Under the system, threat, and network models defined above, we propose a set of properties that the collective system actors known as the Harmonia system need to achieve (given the presented set of assumptions):

- On-chain **Safety**, $\mathcal{G}_1$: no invalid light client updates are validated.
- On-chain **Decentralization**, $\mathcal{G}_2$: anyone should be able to run an application relayer and a SNARK relayer. While bringing the decentralization advantages, it also introduces exploitation vectors, such as the extraction of value (MEV) [55], which we predict will be enabled for the cross-chain scenario soon [14], [56]–[58].
- On-chain **Finality**, $\mathcal{G}_3$: once a message has been delivered and validated, it cannot be reverted.
- On-chain Off-chain **Liveness**, $\mathcal{G}_4$: valid updates are eventually accepted on-chain.
- On-chain Off-chain **Extensibility**, $\mathcal{G}_5$: refers to supporting execution environments other than the EVM (e.g., WASM). The solution should allow cross-chain use cases to expand seamlessly to other chains and reduce the risks of integrating new deployments.
- On-chain Off-chain **Flexibility**, $\mathcal{G}_6$: it is possible to integrate new blockchains, light client protocols, and SNARK schemes. In the context of SNARK-based bridges, this property is needed because new protocols will continue to emerge, while existing protocols might become defunct or undergo significant changes (e.g., forks).
- Off-chain **Safety (Off-Chain)**, $\mathcal{G}_7$: fake proofs of the construction of a block header on a source blockchain are computationally infeasible (this is given by the specific SNARK protocol with which we instantiate the light client).
- Off-chain **Censorship-Resistance**, $\mathcal{G}_8$: no entity should be able to prevent a valid light client from updating against our system.

Furthermore, we present two important performance metrics:

- On-chain **Cost**: the system shall minimize transaction fees and hardware expenses as much as possible, as they are a significant constraining factor.
- On-chain Off-chain **Latency**: the system shall be able to provably verify Ethereum’s state in a reasonable amount of time.

D. Architecture

Figure 2 presents the architecture of Harmonia. The system is based on four building blocks:

1) Circuit generation block (steps 1-6): circuit generation components (steps to create and compile the circuit implementing the light client protocol). The circuit building block deals with the definition of the light client protocol as a set of circuits. A trusted setup ceremony is run, and the respective proving keys and circuits are generated. This is an expensive process that happens only once per circuit version.
2) SNARK generation and relaying (steps 7-11): SNARK generation components (fetching and giving inputs to the circuit, retrieving SNARK). In these steps, the SNARK relayer fetches the necessary information from the blockchain and inputs it into the circuit. This yields a valid SNARK for a specific input.

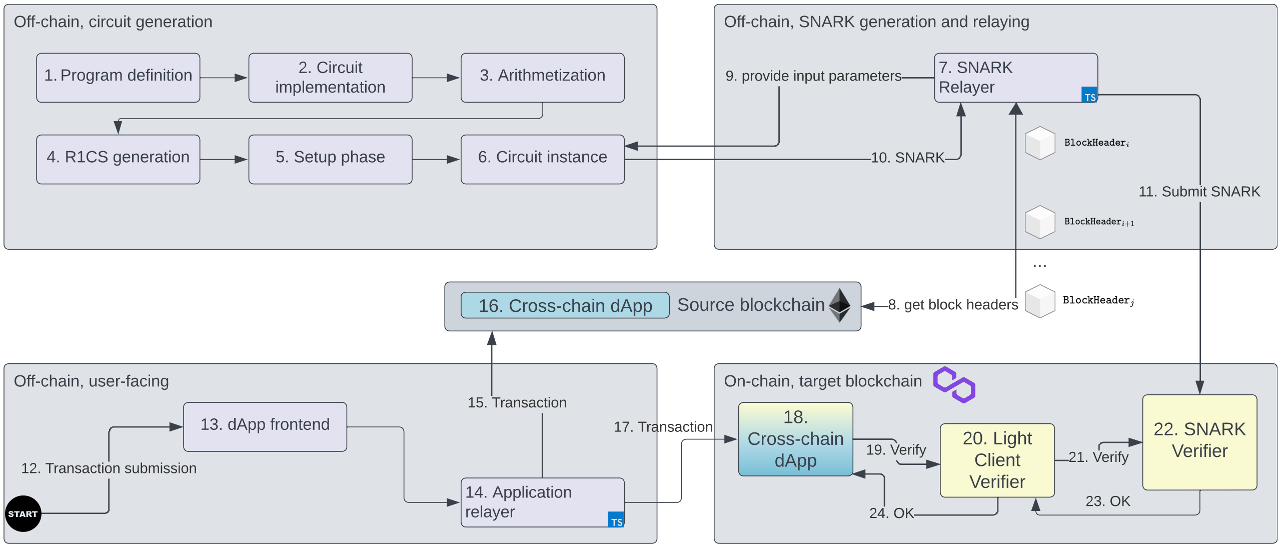


Fig. 2: Architecture of Harmonia. Off-chain components are represented by 🟪 . Destination chain components 🟨 . The cross-chain contract is represented by 🟦 .

3) User-facing application (steps 12-16): user-facing side of the cross-chain dApp (logic on the source chain). Here, we define the logic of the source chain application. A user can interact with it through a user interface connected to the application relayer. The latter transacts against the source chain and fetches a Merkle proof for that transaction.
4) Destination chain cross-chain dApp (steps 17-23): cross-chain logic on the destination chain (logic on the target chain, based on proven facts on the source chain, via a SNARK). In the last steps, we prove facts on the source blockchain (e.g., a transaction on 16 was inserted in a block). The application relayer provides a Merkle proof and data in 17 (via a blockchain transaction). Step 20 validates that Merkle proof against a validated block header (namely, the execution state root that the SNARK verifier provides). After verification, the logic on 18 can be executed, with validated input from 17.

Contracts 16, 18,20, and 22 are deployed once per version (possible to deploy behind a smart contract proxy) by the set of administrators running those contracts (e.g., via a multisig address). The proposed architecture provides a reliable means to authenticate data in Ethereum. The attributes in Ethereum beacon chain block headers reference a "BeaconState" root hash, which points to a recent execution layer block header. The execution layer block header references the root hash of the execution layer state. Thus, if a chain of proofs is also supplied and verified against the light client contract state, it can be used to prove in the targeted blockchain the occurrence of any event in the Ethereum world starting from a Beacon block header, allowing users to build cross-chain applications using Ethereum's state.

E. Altair Light Client 1.0

This section presents the original Ethereum's light client proposal, published in the Altair fork. The Altair Light Client (ALC) relies on a sync committee mechanism, called Altair syncing protocol [59], deployed in the Altair hard fork [60]. Altair enables light clients to efficiently and securely construct the chain of beacon block headers. We discuss the low overhead for light clients, making the beacon chain light client-friendly for resource-constrained environments. After that, we present its limitations.

The algorithm works as follows. Figure 3 presents the process of a light client tracking the chain of recent beacon block headers. The process starts at the current block header. In this figure, a light client has a block header at slot s, with a well-defined state root ① in period i. Let us consider current period i, current slot s, a sync committee at period i formed by 512 nodes $(N_1^i, N_2^i, ...N_{512}^i) = N_{[1:512]}^i = \mathcal{C}^i$. Each node has a private key $K_K^{N_i}$ and public key $K_P^{N_i}$ Let the block header at period i be `BlockHeader`$_i$.

The light client stores the current header so that the light client can authenticate information such as transactions and balances against the header (via Merkle proofs). The light client also stores the current and the next sync committees, obtainable via the state root ②. Using a Merkle proof, it verifies the next sync commmittee in the slot s post-state. This sync committee will sign block headers during period $P+1$. After this, the protocol obtains the public keys of the light client sync committee ③. The keys of the nodes that participated in the aggregate signature of the sync committee are defined in a participation bit map ④. More concretely, this step creates a combined public key $K_P^{N_1,\ldots,N_{512}}$ from the individual public keys of the sync committee subset that participated in the aggregate signature $\sigma_{N_1,\ldots,N_{512}}$. An aggregated signature σ_{1-N} is calculated from the combined public key.

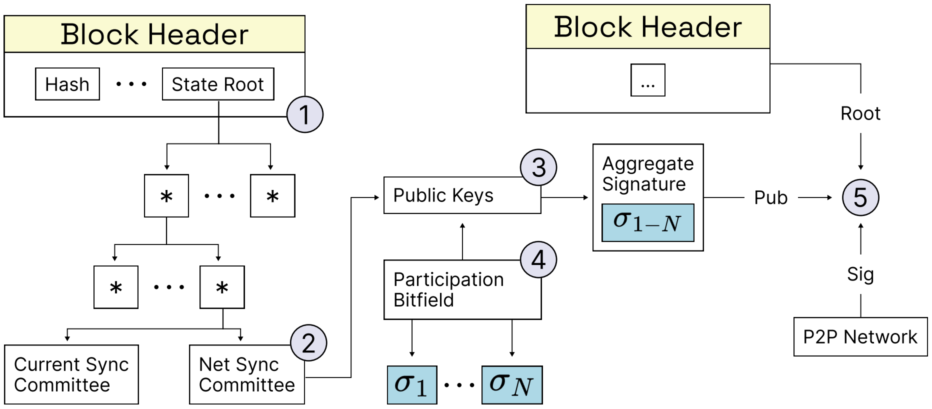


Fig. 3: Light Client Update algorithm (`LCU`) in Altair [59]

That signature (pub) is compared with the aggregate signature downloaded from the block or the peer-to-peer network (sig). Lastly, the signature is verified against the combined public key and the newer block header, $\texttt{VERIFY}_{pub}(root, sig)$, in ⑤.

If the verification passes, the new block header has been successfully authenticated. The next sync committee will become the current sync committee of the next valid period (and, consequently, block) so that light client updates for the following block can be done in a chain. Taking advantage of this property, one can verify that blocks have been validated retroactively. The light client executes the light client update algorithm `LCU` to authenticate a block header in the following period. We formally define this process in Figure 4.

Nevertheless, critical security issues have been identified with Altair [61], [62]. We will review two critical issues with the specification and provide the first solution to this problem. First, the sync committee is not held accountable for misbehaving since slashing is not enforced when sync committee members sign semantically invalid block headers. This incentivizes signers to blindly sign every block header (invalid or not) to reap the rewards, resulting in the loss of safety of the light client. Secondly, a light client sync committee might receive valid attestations but choose not to sign them, breaking the light client's liveness. Therefore, Altair 1.0 does not assure the generated SNARKS are semantically valid, allowing replay attacks and potential misuse.

F. DendrETH: Strengthening the Security of ALC

DendrETH is a smart contract implementation of ALC, which allows one to prove Ethereum facts on target blockchains. To motivate DendrETH, we formalize a new attack on ALC's sync committee, which explores its current lack of accountability, solving technical challenge C_3.This attack is executed by bribing honest nodes and forcing them to generate valid proof for an invalid checkpoint (set of blocks up to a certain epoch). Hence, we call this attack *Ghost Checkpoint Attestation Attack.*

1) Threat Model: As the size of the sync committee is fixed for every 256 epochs, the time window for an attack to corrupt the committee is 512 epochs (54.6 hours). The economic security threshold k will be bounded, considering that each validator needs to stake 32 Eth. Since an adversary needs to control two-thirds of the sync committee to sign block headers, the number of nodes necessary to control would be 341 (collateral slashed would be approximately 20M USD). The following inequality can express this: $k > 512 \times 32 \times 1850$ USD/ETH $\times \frac{2}{3}$. At the current market rate, $k > 10,923$ ETH ($20,207,000$ USD). On the other hand, an adversary would have a time window of $54.6 + 27.3 \approx 82$ hours (nodes in the sync committee are notified 512 epochs in advance + 256 epochs where they operate) to corrupt two-thirds of this committee, where the capital necessary for it could be substantially lower. When the sum protected by the bridge is inferior to k, bribing the committee is not economically secure (assuming a slashing of 100%). However, the number of secured assets (ERC-20, NFTs) is often hundreds of millions of dollars, as we have seen by recently hacked bridges [12]. Note that slashing is proportional to the amount of stake performing the attack. Therefore, we provide an upper bound of the capital needed for an attack. A more lenient estimate shows that if the entire sync committee could be fully slashed down to 0 ETH, this would still cap the security level to the whole stake of the sync committee, or 16384 ETH $\approx$ 32 million USD. We put forward a slashing proposal and a future research direction that together increase the crypto-economic security of the system.

2) Ghost Checkpoint Attestation Attack: The idea behind this attack is that entities in the sync committee can create multiple valid histories of the Ethereum blockchain, sign them, and propagate them without being penalized. When the sync committee creates a valid block and submits it to the network of the targeted domain, there are no safety violations. However, a deceptive supermajority within the sync committee can mislead applications that depend on Ethereum's light client synchronization protocol into accepting a non-canonical

Altair Light Client (ALC) Specification

One initializes the light client $\mathcal{L}$ with the pre-agreed block at period i, $\texttt{INIT}(\texttt{BlockHeader}_i)$ (details in [39]) and obtains the current light client state. The light client state $\mathcal{L}_{\mathcal{S}_i}$ has the following attributes: $\mathcal{L}_{\mathcal{S}_i} \doteq (\texttt{BlockHeader}_i, \mathcal{C}_i, \mathcal{C}_{i+1}, \texttt{F-BlockHeader}_i)$. We omit some variables for brevity, e.g., `best valid update`. The process of doing an update requires the current light client state $\mathcal{L}_{\mathcal{S}_i}$ and auxiliary data for a light client update `LCU`. The light client update data can be obtained from full nodes (namely from a Beacon block header). $\texttt{LCU}_{data}$ is:

1) $\texttt{BlockHeader}_i$: the attested header.
2) $\mathcal{C}_{i+1}$: the next sync committee.
3) $\pi_{\mathcal{C}_{i+1}}$: the *next_sync_committee_branch* is the Merkle path that authenticates the next sync committee.
4) $\texttt{F-BlockHeader}_i$: the `finality_header` is the header whose signature is being verified.
5) $\pi_{\texttt{F-BlockHeader}_{i+1}}$: the Merkle proof that authenticates that the *finality_header* is the header corresponding to the finalized root saved in the *finality_header*.
6) *sync_committee_bits*: a bitmask showing who participated in the sync committee. Allows to calculate which keys will be included to create the aggregated public key.
7) $\sigma_{N_{[1:512]}}$: the aggregated signature of the nodes participating in the sync committee, *sync_committee_signature*.

Light Client Update Algorithm Details

The light client update (`LCU`) algorithm is in Algorithm 7. The **assert** keyword evaluates a predicate, and keeps the execution going if its true, or returns the execution of the program with $\perp$ if the predicate evaluates to false. There are procedures to update both the attested header and the optimistic header. We focus on updating the attested header since both procedures are similar.

```
Algorithm 1: LCU algorithm in Altair 1.0
Input: $\mathcal{L}_{\mathcal{S}_i}$, $\texttt{LCU}_{data}$
Input: slot $s$, genesis root $r$
Output: $\mathcal{L}_{\mathcal{S}_{i+1}}$
1 assert $\texttt{ValidUpdate}(\mathcal{L}_{\mathcal{S}_i}, \texttt{LCU}_{data}, s, r)$ ▷ Appendix F
2 $\texttt{BestUpdate}(\mathcal{L}_{\mathcal{S}_i}, \texttt{LCU}_{data})$
3 $\texttt{UpdateOptimisticHeader}(\mathcal{L}_{\mathcal{S}_i}, \texttt{LCU}_{data})$
4 assert $3 \times \sum \texttt{LCU}_{data}.scb \geq 2 \times |\texttt{LCU}_{data}.scb|$
5 assert $\nexists \mathcal{L}_{\mathcal{S}_i}.\mathcal{C}_{i+1} \wedge \exists \texttt{LCU}_{data}.\mathcal{C}_{i+1}$
6 $\mathcal{L}_{\mathcal{S}_{i+1}} \leftarrow \texttt{ApplyUpdate}(\mathcal{L}_{\mathcal{S}_i}, \texttt{LCU}_{data})$
7 return $\mathcal{L}_{\mathcal{S}_{i+1}}$
```

- Step 1 validates the correctness of the light client update (e.g., calculates slot, asserts validity of headers, asserts freshness of slots, etc - see Appendix F for details and a formal description).
- Step 2 checks which is the light client updates is the best (e.g., which one has a higher supermajority, finality, slot age). This accounts for the case we have to force-update to it if the timeout elapses [39].
- Step 3 updates the optimistic header. It checks that the update is backed by a significant portion of the committee and that the light client is progressing (in terms of the slot). This sets the light client optimistic header to be equal to the update of the attested header, i.e., $\mathcal{L}_{\mathcal{S}_i}.\texttt{F-BlockHeader}_i = \texttt{LCU}_{data}.\texttt{BlockHeader}$. In other words, the finalized header is being updated recurrently with what is thought to be the current finalized header at a given time (but may change until a finalized header is found).
- Step 4 checks that a supermajority signed the update. This is a necessary condition for a header to become finalized.
- Step 5 verifies the next sync committee is defined in the update. Note: a few other checks are done, namely it is determined if an update not only progresses the finalized checkpoint but might also finalize the next sync committee.
- Step 6 updates the internal state of the light client based on the updated information. It increments the sync committee period, updates the beacon slot, updates the current and next sync committee, and updates the finalized header [39].

Fig. 4: Altair Protocol (`LCU` algorithm)

(but valid) finalized header (this is, a proof pointing to a syntactically valid block header that is not included in the canonical chain). For instance, this message could be leveraged to compromise a bridge contract based on the light client sync protocol, reducing its trustworthiness. We describe a specific attack on SNARK-based bridges depending on Altair 1.0:

1) a corrupted subset of the sync committee creates an invalid block.
2) it creates a SNARK for that invalid block.
3) the committee will attempt to delete evidence of their misbehavior and the invalid block is deleted to prevent accountability.
4) the (valid) SNARK attesting an invalid block is sent as proof to the destination chain.
5) arbitrary cross-chain logic is executed based on a non-canonical state.

Since the destination chain has no observability on the source chain by design (otherwise, it would not be a light client), there is no way for it to know the canonical block for this attack. The malicious sync committee can also perform an attack on the liveness of the protocol by synchronizing themselves (à la Flashbots [63]) not to attest block headers, effectively conducting a Denial of Service attack - this would have a cost of ≈ 51.2 ETH $\approx 99,300$ USD per sync period.

3) Sync Committee Slashing: The core of DendrETH's proposal is to make the sync committee accountable by creating and sharing evidence that can lead to slashing. The idea is if the sync committee signs and submits an alternative finalized history, the entire sync committee gets slashed. Only malicious behavior should be slashable - "but a validator tricked into syncing to an incorrect checkpoint should not be slashable even though it is participating on a non-canonical chain. Note that a slashing must be verifiable even without access to history, e.g., by a checkpoint synced beacon node" [64]. DendrETH's `slashing` algorithm is defined in Figure 5. The steps of `IdenfitySlashing` (see figure) are:

- Step 2 asserts that evidence refers to a point in the past.
- Step 3 asserts that the periods in which the conflicting evidence is provided are sequential. This aims to avoid over-penalization (e.g., due to operational issues; also, it may not make sense to penalize validators who misbehaved long ago). Moreover, it allows a validator synced to a malicious checkpoint to contribute again in a future period.
- Step 5 shows the first slashing condition: there are conflicting block headers submitted in the same slot, slashing is applied.
- Step 8 checks that the sync committee did not sign conflicting finalized $\mathcal{C}_{i+1}$. A few checks are omitted for brevity.
- After that, we check the linearity of the finalized block history. Validators are not allowed to sign blocks that suggest a finalized block history that does not follow a linear progression, i.e., $\exists \texttt{HeaderChain}^{i}_{j} \implies \forall k : i \leq k \leq j : \exists \texttt{BlockHeader}_{k}$.
- Step 10 outputs a boolean `linear`[0] (and `linear`[1] respectively).
- Step 12 checks that if the evidence lacks finality, then the recently finalized block root must be the default root.
- Step 14 assumes the finalized history is linear, i.e., a mismatch between $\texttt{BlockHeader}_{|HeaderChain|}.root$ and $\texttt{evidence}[2].root$ suggests that a validator might have signed off on a block that does not belong to the known finalized chain.
- Step 15 shows a check to discover which evidence follows the canonical chain. The idea is to show that at least one of the evidences show the sync committee supports a valid history, and, at the same time, an invalid history. The variable `linear`[0] (similar to `linear`[1]) checks if the actual finalized block root for `evidence`[0] matches the root of its finalized header. If they match, the finalized header in that evidence represents a linear history. Alternatively, if evidence about non-linearity cannot be obtained directly from an attack, it can be proven that one of the `BlockHeader` is part of the canonical finalized chain of $\mathcal{S}$.

The steps of `EnforceSlashing` (Algorithm 3) are:

- Step 3 gets the public keys of each appointed member of the slash committee to be slashed. Same for `evidence`[1].
- Step 6 asserts that there is at least one validator to be slashed.
- Step 8 validates the slashing evidence 0 (and evidence[1]). It does a set of asserts: 1) it ensures that the number of participants in the sync committee meets a minimum threshold, 2) verifies a Merkle proof $\mathcal{B}.\texttt{state.verify}(\texttt{evidence}.\pi_{\texttt{evidence.BlockHeader}^*}, \texttt{evidence.BlockHeader}^*\texttt{.root}, \texttt{root})$, 3) verifies that $\mathcal{C}_{i+1}$ exists, 4) verifies a Merkle proof $\mathcal{B}.\texttt{state.verify}(\texttt{evidence}.\pi_{\mathcal{C}_{i+1}}, \texttt{evidence}.root, \texttt{evidence}.\mathcal{C}_{i+1})$, 5) verifies the aggregate signature: $\texttt{VERIFY}_{K^{S}_{p}.\mathcal{N}}(PP, \texttt{evidence}.aggregate.\sigma_{N_{[1:512]}}) \rightarrow 1$, where PP are the public keys of the validators to be slashed (output of `ToSlash`) in step 3.
- Step 9 slashes the malicious members of the sync committee reported if they are slashable.

A few steps must be taken to apply these protections to SNARK-based bridges. The public input of the SNARKS we use in our construct must be augmented to contain the information for slashing. However, adding more public inputs would make the SNARK verification more expensive and could incur data availability issues because the nodes now need to access SNARK information on-chain to be able to execute the slashing protocol. New data availability mechanisms can expose this data with robust incentives [66]. This scheme can be generalized to include the entire validator set instead of solely the sync committee, further increasing the economic security, which we briefly mention in the discussion section. We discuss the rationale for this generalization in Section VI.

In conclusion, having two pieces of evidence provides a way to demonstrate and compare the conflicting actions of a validator. If a validator provides two contradicting pieces of information in a context where such a contradiction should not be possible, then the two pieces of evidence serve as proof of their wrongdoing, making them eligible for slashing.

G. Building Cross-Chain Applications

To simplify the use of Harmonia in different cross-chain applications, we adopt a modular design where we separate the verification logic (Beacon chain light client verifier) from the application logic (e.g., state sync, bridging). The application logic can consume a standard interface from the on-chain verifier that integrates the consumption of verifier block roots and Merkle trees onto its logic flow. The high-level idea is to use a SNARK-based light client as the source of truth of the source blockchain, providing regular, valid block header updates to an interoperability application. The actors involved in the process are those defined in the system model (Section III). In our implementation, we use DendrETH as the light client protocol and the Ethereum Beacon chain as the source blockchain (cf. Section III-D).

In greater detail, we define two sub-protocols that govern a cross-chain application using Harmonia: the SNARK Relayer protocol (involving steps 3 and 4) and the Application Relayer Protocol (steps 1, 2, 5), involving A and E. The former is presented in the Protocol 4 listing.

DendrETH Specification

DendrETH is a superset of ALC. Thus, we do not define again the `LCU` algorithm, but merely the proposed improvements. We define two data structures to use in our protocol. First, the slashing evidence, or `evidence`. Secondly, we define the slashing action $\mathcal{S}$. A slashing starts with a beacon state $\mathcal{B}$ [65] and a slashing action. We represent the action of slashing a set of validators $\mathcal{N}$ by $\mathcal{S} \xrightarrow{\texttt{evidence},\mathcal{N}} \{1\}$

We define `evidence` as having several attributes:

1) `BlockHeader*`: an attested block header, which is the contender that originates the slashing.
2) $\mathcal{C}_{i+1}$: the next sync committee.
3) $\pi_{\mathcal{C}_{i+1}}$: the *next_sync_committee_branch* is the Merkle path that authenticates the next sync committee.
4) `BlockHeader`: the finalized block header.
5) $\pi_{\texttt{BlockHeader}^*}$: a Merkle branch validating the finalized block header.
6) `aggregate`: the aggregated signature $\sigma_{N_{[1:512]}}$ of the nodes participating in the sync committee, *sync_committee_signature* and the bitmap of the participants *bmap*.
7) *slot*: the signature slot.
8) $K_P^{N_1}, \ldots, K_P^{N_{512}}$: the sync committee public keys.
9) *root*: represents the root hash of a block that the evidence claims to be finalized.
10) π_{root}: Merkle proof showing the inclusion of *root* in the state tree.

It is worth noting that there are two evidences provided in a slashing. The purpose of having two distinct shreds of evidence is to provide proof that a particular validator (or set of validators) committed an equivocation or another malicious act. Equivocation essentially means producing multiple conflicting pieces of information for the same context.

The slashing action $\mathcal{S}$ has:

1) $\mathcal{N}$: a list of validators to be slashed.
2) `evidence`.
3) $\texttt{BlockHeader}_{|HeaderChain|}.root$: the recent finalized block header root.
4) *slot*: recent finalized slot.

The complete slashing algorithm has two parts: `IdentifySlashing` and `EnforceSlashing`. The latter should only be called if the former returns 1 (otherwise, it means that the slashing evidence is not valid).

```
Algorithm 2: IdenfitySlashing
Input: $\mathcal{B}, \mathcal{S}$, evidence[2]
Output: $\{\bot, 1\}$
slash ← 0 ▷ slash flag
assert $\mathcal{S}.slot >$ evidence.$slot$
▷ asserts both evidences are sequential
if $evidence[0].slot = evidence[1].slot \land evidence[0].BlockHeader^* = evidence[1].BlockHeader^*$ then
    slash ← 1
end if
if $evidence[0].\mathcal{C}_{i+1} \neq evidence[1].\mathcal{C}_{i+1}$ then
    slash ← 1
end if
linear[0] ← evidence[0].$root$ == evidence[0].BlockHeader.$root$
if $\neg(final(evidence[0]) \lor final(evidence[1]))$ then
    assert $\texttt{BlockHeader}_{|HeaderChain|}.root = \emptyset$
end if
▷ Checks to prevent slashing validators who signed an alternate history non-maliciously
canonical_is_0 ← evidence[0].BlockHeader.$slot \geq$ evidence[1].BlockHeader.$slot \land slot ==$ evidence[0].BlockHeader.$slot \land$ $\texttt{BlockHeader}_{|HeaderChain|}.root ==$ evidence[0].$root \land$ linear[0]
▷ Might check $\mathcal{B}$ and $\mathcal{S}$
if $canonical\_is\_0 \land slash$ then
    return 1
end if
```

```
Algorithm 3: EnforceSlashing
Input: $\mathcal{B}, \mathcal{S}$, evidence[2]
Output: $\{\bot, 1\}$
ToSlash ← $\emptyset$
for key in $\mathcal{S}$.evidence[0].$K_P^{N_1}, \ldots, K_P^{N_{512}}$ do
    ToSlash = ToSlash ∪ GetPublicKey(key, aggregate.bmap)
end for
for val in $\mathcal{S}.\mathcal{N}$ do
    ▷ asserts at least one validator val to be slashed $\in \mathcal{B}.validators$
end for
assert ValidateSlashingEvidence(evidence[0], $\mathcal{S}.\texttt{BlockHeader}_{|HeaderChain|}.root$, $slot$, $\mathcal{B}$.GenesisRoot)
$\mathcal{S} \xrightarrow{\texttt{evidence},\mathcal{N}} \{1\}$
```

Fig. 5: DendrETH specification - `Slashing` algorithm

Protocol 4: SNARK Relayer Protocol

```
Input: Relayer private key $K_k^r$
Input: Full node list $N$, update period $\Delta_{proof}$
Input: Light client verifier contract address $addr_c$
Input: Light client circuit $\mathcal{C}$ and Prover algorithm P
Data: Location of the latest block header, $add_b$
Data: Access to the source and target blockchains $\mathcal{B}_s$, $\mathcal{B}_t$
Result: Sends a SNARK to $\mathcal{B}_t$ that validates validity of a block header from $\mathcal{B}_s$
1  Procedure SendSNARK
2      LCU$_{data}[n] \leftarrow \emptyset$ ▷ array with $n$ values
3      for every $\Delta_{proof}$ do
4          for each $n$ in $N$ do
5              BlockHeader $= n.read_{\mathcal{B}_s}(add_b)$
6              LCU$_{data}[index]$ = GetLCUDataFromBlock(BlockHeader)
7          assert$\forall i, j : 1 \leq i \leq n$, LCU$_{data}[i]$ = LCU$_{data}[j]$ ▷ responses from different nodes are consistent
8          $(x_{\mathcal{C}}, w_{\mathcal{C}})$ = GenerateProofInput(LCU$_{data}$) ▷ Generate input and witness
9          $\pi_{SNARK} = P(\mathcal{C}, x_{\mathcal{C}}, w_{\mathcal{C}})$ ▷ Appendix D
10         $\sigma \leftarrow$ SIGN$_{K_k^r}(\pi_{SNARK})$
11         store$_{\mathcal{B}_t}(add_c, (\pi_{SNARK}, \sigma))$
```

Protocol 5: Application Relayer Protocol

```
Input: Relayer private key $K_k^r$
Input: Cross-chain logic contract address on the source and target chains, $addr_s$ and $add_t$, respectively
Input: State Merkle tree on the source chain $\mathcal{B}_s.tree$
Input: Cross-chain logic $\zeta_l$
Input: Cross-chain current state $\zeta_s$
Data: Access to the source and target blockchains $\mathcal{B}_s$, $\mathcal{B}_t$
Result: Executes one step of cross-chain logic verified by Merkle proofs
1  Procedure UpdateCrossChainContract
2      while true do
3          $tx_s$ = GetTransactionForLogicNow($\zeta_s, \zeta_l$)
4          store$_{\mathcal{B}_s}(addr_s, tx_s)$
5          $\pi \leftarrow \mathcal{B}_s.state$.proof($tx_s$) ▷ gets Merkle proof for transaction
6          $\zeta_s \leftarrow$ UpdateCrossChainState($tx_s, \pi$)
7          $tx_t$ = SIGN$_{K_k^r}$(GetTransactionForLogicNow($\zeta_s, \zeta_l$))
8          store$_{\mathcal{B}_t}(addr_t, (tx_t, \pi))$
9          $\pi' \leftarrow \mathcal{B}_t.state$.proof($tx_t$)
10         $\zeta_s \leftarrow$ UpdateCrossChainState($tx_t, \pi'$)
```

The SNARK relayer protocol runs every time interval defined (e.g., every 32 slots/blocks). To maximize resilience and stemming from the increasing usage of node-as-a-service companies such as Blockdaemon (maximizing efficiency but with security risks), the SNARK relayer should query several sources (line 4). In line 5, the relayer queries a full node and gets the latest block. The latest block contains the information necessary to build LCU$_{data}$ (line 6). In line 7, we assert that the responses from different node providers are the same. Otherwise, we abort the protocol.

In line 8, the SNARK relayer generates the necessary input and witness to be used as input to our light client circuit. In line 9, we generate the SNARK using the Prover algorithm P. It takes as input the light client circuit we developed and inputs a SNARK. The relayer signs the generated data in line 10 for accountability. Finally, in line 11, the SNARK is submitted to the Light Client Verifier contract on the target chain[5]. We note that any node can perform the role of the relayer, making the SNARK submission process permissionless and decentralized. An incentive mechanism can be built to incentivize the good functioning of the network. Details on the on-chain light client verifier can be found in the implementation section.

We now explain the Application Relayer Protocol (Protocol 5). First, the application relayer takes as input the cross-chain state (the next step of the cross-chain logic). In line 3, the relayer creates a transaction that executes the next step on the source blockchain (this transaction is signed, similarly to line 10 of Protocol 4, for accountability, but we omit it for brevity). On line 4, it submits the transaction on the source blockchain. Line 5 generates a Merkle proof proving the inclusion of tx_s in the source blockchain. Line 6 depicts the relayer updating the cross-chain state. After that, the relayer creates a transaction according to the cross-chain logic aimed at the target blockchain. Then, it transacts against the target blockchain, in line 8. Finally, the relayer updates the cross-chain state. The next section presents a use case of this framework.

H. State Migration with Harmonia

This section presents an implementation of an application using Harmonia and DendrETH. Our proof of concept is based on SmartSync [67]. SmartSync implements state migration across EVM-based chains. The migration of state can be performed by interacting with pairs of smart contracts, one in each chain, and retrieving Merkle proofs for the updates on both contracts. Although some preliminary research has been done on the topic [68], [69], it is still not possible to automatically migrate decentralized applications state to blockchains with a different runtime (i.e., migrating state across heterogeneous chains) [21]. We leave the aspect of automatic migration for future work and instead focus on

---

[5] Technically, we send the SNARK and the data we want the smart contract to record. We explain this process in detail in the implementation section.

migrating state across EVM chains (i.e., homogeneous chains), in a trustless and decentralized way using DendrETH.

The starting point is a pair of (equal) smart contracts deployed on two EVM chains. A user or relayer will issue a transaction to one of the contracts and then reissue that same transaction to the other contract. The migration is verified by a third contract upon providing a pair of Merkle proofs (one for each transaction on each contract) and auxiliary data. The rationale is the following: consider that the storage hash is the Merkle root of the storage tree, which encodes a key-value store for each contract. If the contract updates on both ends yield the same storage hash and both Merkle proofs are valid, then both contracts have the exact same state at the block to which the Merkle proof refers. However, verifying Merkle proofs relies on a centralized, trusted relayer to provide the source of truth (block header roots). We remove this dependency by integrating our fork of SmartSync with Harmonia, showcasing a use case for interoperability other than asset transfers. The idea is based on four smart contracts. A cross-chain logic contract on the source chain, a cross-chain logic contract on the target chain, an application proof verifier contract, and a light client verifier contract.

Figure 6 shows the sequence diagram of the use case. In step 1, an application relayer or an end user interacts with the source contract, modifying its storage (update value v). A Merkle proof π for the storage modification is constructed (step 2). After that, the same transaction is made against the logic contract on the target chain (step 4), and a proof π' is generated (step 6). Next, the relayer submits a verification request that validates the migration. It takes as input the two generated Merkle proofs and requires the storage hashes of the contracts to be the same. The application proof verifier contract will interact with the Light client Verifier contract in step 8 to get the latest validated execution state root r, which was validated using a SNARK π_{SNARK}. After that, it will validate both Merkle proofs using the validated block header root. Upon verification, the state migration is complete.

Note that the cross-chain logic does not enforce that an update on the cross-chain logic contract on the target chain is contingent on the state of the source. A relayer could update the target contract at will. However, the verification process would fail if the updates on the target contract do not correspond to the updates on the source contract, as the storage roots would be different. We enforce the three cross-chain rules through the light client and the different application contracts. Formally, we denote our cross-chain logic ζ [12] that depends on local transactions $\mathcal{E}$ happening on the source chain and destination chains, denoted by e^s and e^d, respectively. We will provide implementation details in the next section. Cross-chain logic is then a function of local events. Formally:

$$\zeta(\mathcal{E}) = \begin{cases} \zeta_1(e) = \forall e : e^s \vee e^d & \text{included chains} \\ \zeta_2(e) = e^s \prec e^d & 1^{st} \text{ transaction is in chain } s \\ \zeta_3(e) = e^s.target = e^d.target & \text{we replicate transactions} \end{cases} \tag{1}$$

IV. IMPLEMENTATION

In this section, we provide implementation details on the Harmonia components. We emphasize a variety of technical challenges for our implementation:

1) **Parsing and processing blockchain data**: Although blockchain data is publicly accessible, parsing this data and extracting the required information is complex. Specialized tools and techniques are required to handle this data effectively. For example, the increasing complexity of the Ethereum blockchain, particularly in its need to distinguish between the execution and consensus layers of Ethereum – each presenting unique challenges —- calls for the separate parsing of each layer before interconnecting them. We parse blockchain data from different sources using Blockdaemon's Universal API, a wrapper over RPC nodes of different protocols. This allows us to retrieve data in a uniform format that we can use for our use case. This API also enables us to parse data from other EVM chains effectively.
2) **Engineering effort for site reliability engineering in data collection**: we need to account for the relayers to be fully operational and the blockchain nodes we use to gather necessary data for proofs and cross-chain logic execution. Deploying and maintaining nodes in heterogeneous interfaces is a recognized difficult task, which is the basis for several companies in the space [70]–[73]. We leveraged several institutional-grade node providers to collect data reliably.
3) **Handling different data finality models**: this issue has to do with data reliability: on probabilistic consensus blockchains, forks happen. A challenge is ensuring the on-chain data collected is up-to-date and final (depending on the specific cross-chain logic). Therefore, there is a trade-off between liveness and safety that is not trivial to optimize. In our proof of concept, we choose safety over liveness and await transaction finalization. The application relayer operates with a delay to provide a safe buffer waiting for the source chain's finalization.
4) **System Complexity**: Harmonia comprises several decentralized systems that cooperate asynchronously. In particular, the developed circuits are based on state-of-the-art cryptography and technology, more susceptible to bugs that are hard to identify than stable technologies [14]. A careful security-first implementation must be made to minimize risks and attack surfaces. However, we recognize that our technology must be audited before production deployment.

A. SNARK Relayer

In this section, we describe the SNARK Relayer (SR) implementation. The relayer efficiently generates proofs and publishes updates for all blockchains supported by Harmonia. We provide up-to-date Docker images and Nix environment configurations to simplify the process of running a relay. Our relayer has comprehensive setup instructions and is open

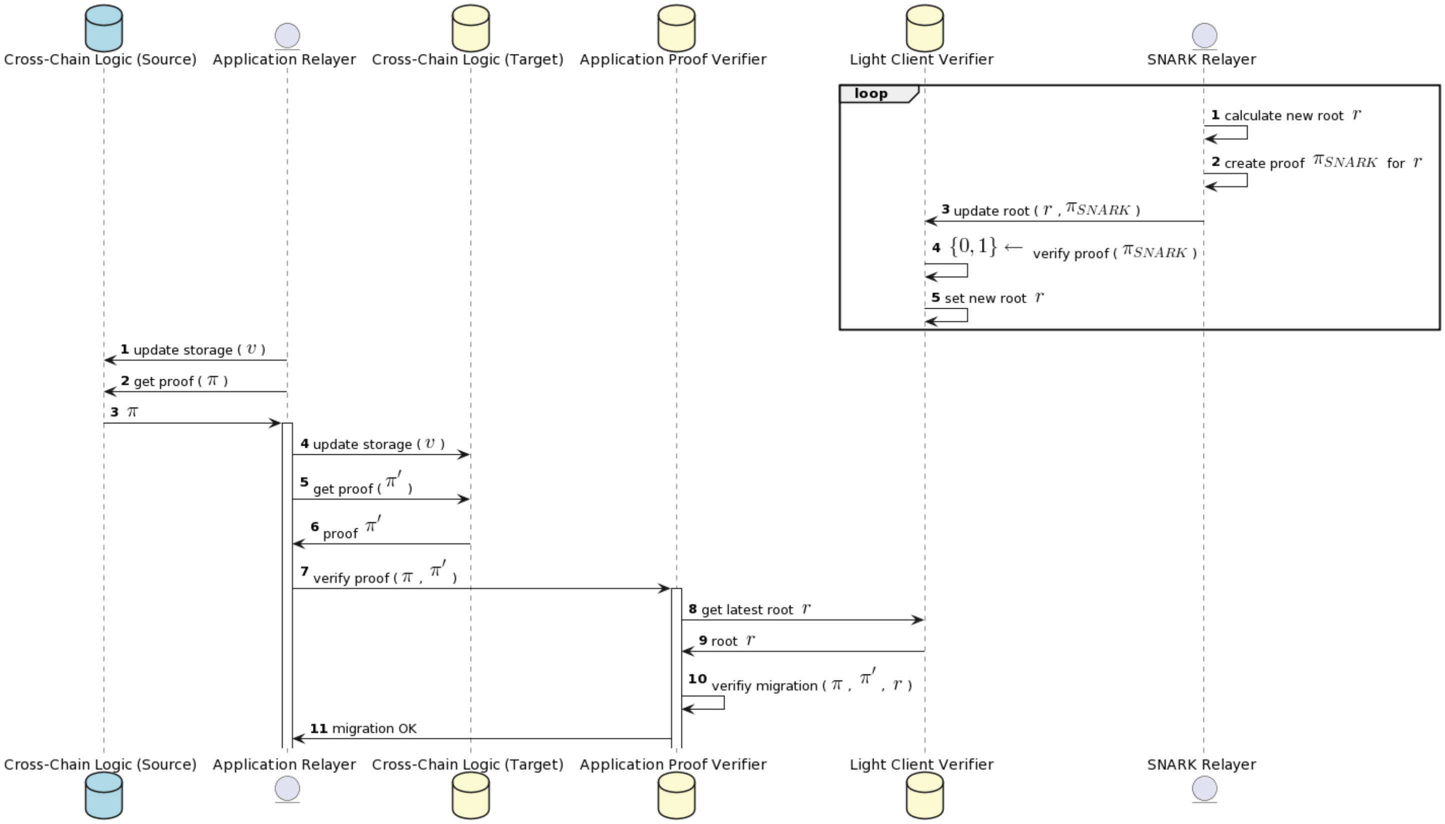

Cross-Chain Logic (Source)
Application Relayer
Cross-Chain Logic (Target)
Application Proof Verifier
Light Client Verifier
SNARK Relayer
loop
1 calculate new root r
2 create proof π_{SNARK} for r
3 update root (r , π_{SNARK})
4 $\{0,1\} \leftarrow$ verify proof (π_{SNARK})
5 set new root r
1 update storage (v)
2 get proof (π)
3 π
4 update storage (v)
5 get proof (π')
6 proof π'
7 verify proof (π , π')
8 get latest root r
9 root r
10 verifiy migration (π , π' , r)
11 migration OK

Fig. 6: Sequence flow of our fork of SmartSync integrated with DendrETH, using Harmnoia. The components of SmartSync are the cross-chain logic (source and target). The blue component is a contract deployed on the source chain. Yellow components are contracts deployed on the target chain. Purple components are off-chain components.

source[6]. The initial slot for the relayer to start creating SNARKs is customizable by setting the finalization time interval `SLOTS_JUMP` (by default 64, i.e., 12.8 minutes). This interval will set a trade-off between liveness and cost: the shorter the interval, the more live the updates are (up to one update per epoch). However, a higher frequency of updates requires a higher workload for creating SNARKs and submitting them on-chain, raising operational costs. The relayer is implemented in Typescript (using NodeJS as the runtime environment) and is composed of multiple workers. Figure 7 illustrates the architecture of the relayer. Workers are implemented using BullMQ, a queue system built on NodeJS. The workers are:

- Update Polling Worker: The relayer itself executes a recurring job that repeats every `SLOTS_JUMP` and starts from `INITIAL_SLOT`. The job targets a specific source chain supporting Pratter (Goerli) and the Ethereum Mainnet. It retrieves $\texttt{LCU}_{data}$ from the Beacon REST API and saves the last downloaded light client update for the job in a Redis database. The data includes the block header at slot `INITIAL_SLOT` and block header at slot `INITIAL_SLOT` + `SLOT_JUMP`, and the necessary parameters for the witness generation consumed by the next worker.
- Proof Generation Worker: After that, the Proof Generation Worker puts the update on a queue that is consumed by the Prover Server (a wrapper over a rapidsnark server[7]), using input from the Update Polling task. The generation worker generates a SNARK and public input to the verifier smart contract and returns the output to the relayer, which persists the proof. The Prover Server requires a path to a build folder containing a `light_client.zkey` and a `light_client.dat` files. The .zkey files are associated with the proving and verifying keys of the proof system instantiation. After a circuit is compiled with Circom, a setup phase is performed where the proving and verifying keys are created, typically in the form of .zkey files.

The prover server runs SNARKJs [74], where an initialization procedure occurs. The procedure includes performing a “Powers of Tau” ceremony[8] at, compiling the circuit, conducting the setup with Groth16 (with the BLS12-381 curve), giving as input the witness received by the proof generation worker,

---

[6] see the relayer code, https://tinyurl.com/2hnauda6.

[7] see rapidsnark, https://github.com/iden3/rapidsnark.

[8] using the public .tau files available in the snarkjs repository, https://github.com/iden3/snarkjs.

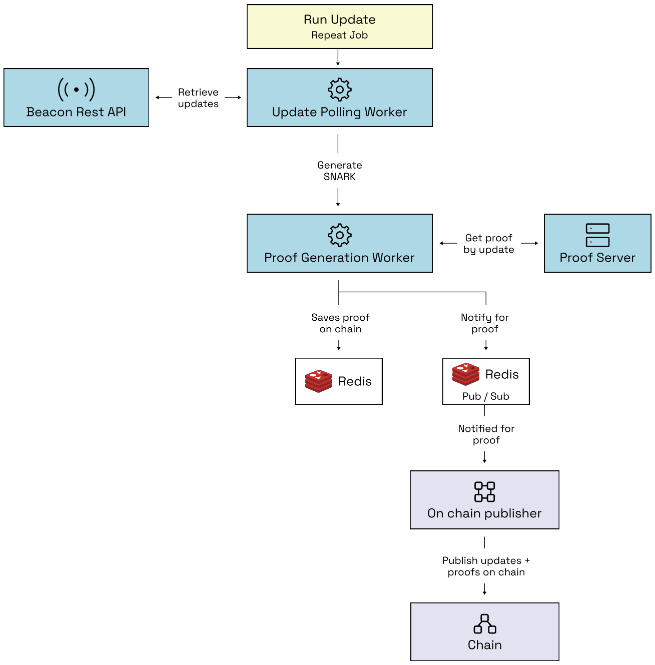


Fig. 7: SNARK Relayer architecture.

and creating the proof. Upon completion of proof generation, the generated proof is saved in Redis. Multiple instances subscribing to this notification attempt to publish the proof on-chain to the verifier contract[9]). Due to the standardized light client verifier interface, the relay architecture allows for extensibility and includes different chains and transaction types. The three workers have been implemented in $\approx 2,400$ lines of code (LOC) of Typescript.

B. Cross-Chain Logic

The cross-chain logic contracts for our PoC are implemented using Solidity: a SimpleStorage contract, a Relay Contract, a Proxy Contract, and utility contracts. The SimpleStorage contract, along with the Proxy, implements the cross-chain logic. The Relay contract verifies Merkle proofs done against the Light Client Verifier contract. We utilized Foundry, a smart contract framework [75] to implement the integration with Harmonia instantiated with DendrETH ($\approx$ 1,000 LOC of Solidity) and test our contracts ($\approx$ 300 LOC of Solidity). We have made our open-source implementation available[10].

C. Application Relayer

The application relayer is an off-chain server that has a dependency on our fork of SmartSync[11]. It receives calls from an end-user via a user interface and executes cross-chain logic: issue transactions to change the state on the source blockchain (e.g., storage of variable "A" of the cross-chain logic contract), fetches recent blocks from the source and destination chains, adding blocks to the relayer contract, creates Merkle proofs from on-chain transactions, obtains latest proofs and input from DendrETH, and conducts the state migration the overall flow is depicted in Figure 1. It contains an OpenAPI specification that generates the SDKs in different programming languages so that different stacks can use the application relayer. The relayer is parameterized with a wallet (so it can transact against the source and destination chains), node RPC endpoints for Ethereum Goerli and Polygon Mumbai and the addresses of the cross-chain logic contracts, the relay contract, and DendrETH.The relayer has been implemented in $\approx 2,150$ LOC of Typescript. Our fork of SmartSync has $\approx 1,500$ LOC of Typescript.

D. Light Client Verifier & Application Proof Verifier Contracts

In this section, we elaborate on the implementation details of the on-chain contracts for validating SNARKs and implementing the cross-chain logic.

1) Verifiers For EVM-based chains: The light client verifier contract has been implemented in Solidity for EVM-based chains (Ethereum Classic, Binance Smart Chain, Polygon, Avalanche, Celo, Theta, Hedera, Fantom). It is divided into the Beacon light client contract and the light client update verifier (SNARK verifier). The light client verifier exposes multiple public functions that consume the outcome of the verifying process, e.g., optimistic headers and execution state roots.

The light client verifier calls an auxiliary contract (SNARK Verifier) generated by SNARKJs, allowing us to verify SNARKS on-chain. The SNARK verifier uses a pairing library that provides functions for performing operations on elliptic curves. It provides functions for addition and scalar multiplication of points, the negation of points, and the bilinear map and pairing operations. The main function of our light client verifier is called `verifyProof`. It takes as the input a SNARK π_{SNARK} (parameters a,b,c that represent points in an elliptic curve) and the *public inputs*, that is, the data to be made available to be consumed by other applications (namely execution state root, optimistic header root, the attested header root, and the attested header slot). Listing 1 shows the code for the verifier contract. The contract computes a commitment to the input of the SNARK on-chain and passes that along to the SNARK verifier and SNARK proof.

```
pragma solidity 0.8.9;
import './Verifier.sol';

contract LightClientUpdateVerifier is Verifier {
  function verifyUpdate(
    uint256[2] memory a,
    uint256[2][2] memory b,
    uint256[2] memory c,
    bytes32 prevHeaderHash,
    bytes32 nextHeaderHash,
    uint256 nextHeaderSlot,
    bytes32 finalizedHeaderRoot,
    bytes32 executionStateRoot,
    bytes32 domain
  ) internal view returns (bool) {
```

---

[9] an example of a successful update, https://tinyurl.com/4saa55fx.
[10] at Github, https://github.com/RafaelAPB/data-transfer-dendreth.
[11] implementation available here, https://github.com/RafaelAPB/smart-sync.

```
16      bytes memory concatenated = abi.encodePacked(
        prevHeaderHash, nextHeaderHash,
        finalizedHeaderRoot, executionStateRoot,
        nextHeaderSlot, domain);
17      bytes32 commitment = sha256(concatenated);
18
19      uint256[2] memory input;
20
21      input[0] = (uint256(commitment) & (((1 << 253) -
         1) << 3)) >> 3;
22      input[1] = (uint256(commitment) & ((1 << 3) - 1)
        );
23
24      return verifyProof(a, b, c, input);
25    }
26  }
```

Listing 1: LightClientUpdateVerifier contract

2) Verifiers for non-EVM-based chains: For blockchains with a WebAssembly runtime, we developed a direct implementation of the light client syncing protocol based on the highly efficient BLS, SSZ, and Light client syncing libraries developed by Supranational and the Nimbus team. We do this by compiling Nim code that implements the light client syncing protocol to C, and then C code to WASM[12].

Furthermore, we provide two codebases we believe to be useful for researchers and practitioners: 1) a direct implementation of the light client protocol, which we adapted to run as a CosmWasm smart contract, and 2) a SNARK verifier in WASM which we also adapted to run as a CosmWasm smart contract.

E. Circuits

We implemented the circuits for DendrETH (without slashing), in 3,600 LOC. We used the Circom programming language [76], snarkJS, and the Groth16 proof system to generate our SNARKs. We chose Circom because it is the most production-ready language that compiles circuit descriptions into arithmetic circuits. The main circuit, `light_client.circom`[13], uses 12 additional auxiliary circuits (e.g., to compute the domain, verify a Merkle proof, calculate a supermajority).

Technically, the circuit is a Circom template that takes two arguments. The first is the expected number of the light sync committee (N). The second argument (K) represents the number of field elements used in the BLS aggregate signature. These arguments help accommodate committees of variable sizes and different signature representations. The circuit receives signals (variables of the circuit), which correspond to $\texttt{LCU}_{data}$. The SSZ (simple serialize) representation, a serialization standard used in Ethereum, is used in some variables (e.g., `prevHeaderFinalizedSlot`), to facilitate its usage in Merkle proof verification. The circuit calculates the target domain (see Section IV-E1), the BLS aggregate signature verification, and a commitment. The commitment includes different pieces of data combined and hashed to summarize the inputs (previous header hash, next header hash, finalized header root, execution state root, next header slot bits, domain). This output commitment is a succinct representation of the data processed and is used to verify on-chain that the created SNARK has an associated input. The circuits currently support generating proofs for a valid light client update.

1) Target Domains: We use the `domain` variable (derived from hardcoded metadata from the network it refers to) to allow the sync committee to identify messages from different systems and assign a purpose. This prevents a new type of replay attack where an attacker requests a signature over a message from an honest user. While that message may have a particular context in a network, its consequences might differ in another network - a light client would accept it because the signature looks correct (domain apart). For example, consider a transfer transaction signed for chain A but later used in chains A and B, effectively attempting to steal user funds in chain B. Setting a domain prevents this attack. The domain definition depends on the domain sync committee, the fork version, and the genesis validator's root.

2) Committment: The idea of our scheme is that when one validates a SNARK, the data passed along that transaction is authenticated and can be used. However, a malicious user can send a valid SNARK to the light client verifier contract and fake data. The way to mitigate this attack is by using a commitment. The commitment binds variables that are public inputs to the circuit. In this way, the light client verifier contract takes a commitment as input that is validated before validating input data (see lines 21 and 22 of Listing 1).

F. Trusted Ceremony

We conducted a trusted setup in two phases. The first phase ("Powers of Tau") is a multiparty computation protocol that constructs a public parameter (CRS) for all SNARK proofs within a certain circuit size. We used the $pot25\_final.ptau$ powers of tau file (which works with circuits up to 2^{25} constraints[14] from the snarkjs library and the $BLS12-381$ curve. The second phase needs to be executed for each subcircuit. Here, the participants use the Powers of Tau phase output to generate a structured reference string by calculating the encrypted evaluation of the Lagrange polynomials at tau. It takes the beacon *ptau* file generated in the previous step, and outputs a final *ptau* file, which will be used to generate the circuit proving and verification keys. Together, the two phases constitute the complete trusted setup ceremony necessary for secure usage of the Groth16 SNARK protocol. This trusted setup is meant for testing purposes.

G. SNARK Generation

First, we compile the circom circuits using the circom compiler. The output is a R1CS representation of the circuit. Then, we use snarkJS to generate the witnesses for input to the light client circuits, corresponding to a light client update. Additionally, snarkJS takes a verification key (generated in the trusted setup phase) as input to generate a SNARK.

---

[12] see proof of concept, https://tinyurl.com/bp73sfmr.

[13] available in the DendrETH repository, https://tinyurl.com/mtzutesj.

[14] available via the hermez project, https://tinyurl.com/534vva2x.

V. EVALUATION

This section evaluates the latency, throughput, and cost of the SNARK Relayer, the Application Relayer, the Light Client Verifier, and the Application Proof Verifier. Let us recall our performance goals: the system tries to minimize costs as much as possible, and it should be able to verify Ethereum's state in a reasonable amount of time. The exchange rates and gas prices are as of 14 July 2023.

A. Setup

We have launched several nodes to support connectivity to the blockchains we connect to. For the consensus layer of the Goerli/Prater network, we launched a Nimbus Client [77] paired with Geth v1.11.5 [78] and downloaded the Prater blockchain (around 60 GB). For Polygon Mumbai, we launched a node v0.3.7 [79]. The size of the blockchain is 260 GB. Both nodes are located in Amsterdam, Europe. When connection to other blockchains was needed, we leveraged the infrastructure provided by Blockdaemon to connect to Ethereum and Polygon. For the SNARK relayer, we deployed a server with 384 GB of RAM, 32-core, 1TB NVMe hard drive, configured with 500GB of swap space and an i9-13900 CPU. The Application Relayer was deployed on a 16 GB RAM, 1 TB SSD, and a ten-core 3.2GHz laptop.

B. Circuits

Our main circuit (`light_client.circom`) took 6 hours, 27 minutes, and 47 seconds to compile on the specified hardware, and the trusted setup phase took 26 hours, outputting a proving key of 55.6GB. It has 410 template instances, $\approx$ 90 million non-linear constraints, $\approx$ 5 million linear constraints, 0 public inputs, 2 public outputs, 20,961 private inputs, 0 private outputs, $\approx$ 93 million wires, and $\approx$ 470 million labels, being one of the most complex Circom circuits developed to date. We developed a test suite using snarkit2 [80] to evaluate the correctness of the sub-circuits that `light_client.circom` uses. For example, our tests for the `pow` circuit have five cases from which we illustrate two: 1) on input `base: 10, power: 3`, the output should be `1000`; 2) on input `base: 2, power: 10`, the output should be `1024`. Table III (Appendix G) showcases the circuit test results, namely the number of template instances, non-linear and linear constraints, public inputs, public outputs, private inputs, private outputs, wires, and labels.

C. Latency

There are two latencies that we are interested in measuring. First, the latency of generating a SNARK proof (Δ_{proof}) to be submitted and validated on-chain. An attentive reader might notice that such a computationally intensive task can upperbound the total latency of our system, but we show this is not the case - instead, finality is the main responsible for latency. Secondly, we want to measure the latency of executing the cross-chain logic of our use case (represented by Δ_{ζ}). These two latencies add up to the total latency, or end-to-end latency (Δ_{total}) for a fact to be verified on a target chain, using Harmonia - this shows how applicable our proposal is in the real world. This includes measuring both the SNARK and the Application relayers (Δ_{snark} and Δ_{app}, respectively), issuing transactions against the source chain (Ethereum) and destination chain (Polygon) ($\Delta^{Ethereum}_{\texttt{store}}$ and $\Delta^{Polygon}_{\texttt{store}}$, respectively, note that this includes accommodating finalization times, which range from a few minutes to around twenty minutes). Note that generating a proof is included in the SNARK relayer operations, i.e., $\Delta_{snark} \geq \Delta_{proof}$. It is important to note that some operations depend on others, while some can be parallelized. In particular, the cross-chain logic rules ζ define that transactions on Ethereum happen before the ones in Polygon. Note that in our use case, we have one transaction in the source chain and two transactions on the destination chain. Generalizing for arbitrary cross-chain logic, the specific deltas depend on the numbers of transactions on the source and destination chains ($|tx|_s$ and $|tx|_d$, respectively). The end-to-end latency is given by $\Delta_{total} = \Delta_{snark} + \Delta_{\zeta}$. The latency Δ_{slack} sums the duration of one slot (12 seconds), which is the lag the relayer has with regard to the current block. This is because the current block (head block) aggregates the claims of the sync committee regarding the previous block: in the best-case scenario, we can start creating a SNARK of the penultimate block. Expanding the expression we obtain:

$$\begin{aligned}\Delta_{total} = {} & \Delta_{snark} + \Delta_{slack} + \Delta_{tx_s} \\ & + (|tx|_s \times \Delta^{source}_{\texttt{store}}) \\ & + \Delta_{app} + (|tx|_d \times \Delta^{target}_{\texttt{store}})\end{aligned} \qquad (2)$$

Let us calculate Δ_{snark}. Consider a chain of block headers on the source chain $\texttt{HeaderChain}^j_i$ from block i to j. Let t_i be the time block i was finalized. Let t_j be the time block j was finalized. Let the finalization latency of the cross-chain logic transaction issued by the Application Relayer on the source chain (defined as tx_s) be denoted by Δtx_s. Let i be the block at which the SNARK relayer starts building a proof for block i. Let $j \pm \epsilon$ be the block where the SNARK relayer has finished building a SNARK for block i (ϵ accounts for small delays on the SNARK relayer software, for example, doing API calls and calling internal functions). Let Δ_{proof} be the time between i and j, i.e., the time the SNARK relayer needs to build a SNARK.

Figure 8 presents our latency model. In step ①, the user or application relayer on behalf of the user submits a transaction t_s to the source chain, following rule ζ_2. We have two possibilities: we either issue t_s before (②) or after i (②'). If issued before i, it means that the SNARK relayer will pick such transaction at block i for when it starts building a proof. Otherwise, it will be picked in the following block that will be verified (not every block is verified). The SNARK relayer picks block i, in step ③ to construct a SNARK. At block $j = i+\Delta_{proof}\pm\epsilon$ the tx_s is finalized, in ④. After the SNARK for block i is created, it can be verified in the destination chain, via t_j, in step ⑤, and eventually included in a block, ⑥. After that, the Application Relayer executes the rest of the cross-chain logic by broadcasting tx_d (executing rule ζ_3, in step ⑦). Formally:

$$\Delta_{snark} = \begin{cases} \Delta_{tx_s} + \Delta_{proof} & t_s < t_i \\ (\Delta_{proof} - \Delta_{tx_s}) + \Delta_{proof} & t_s \geq t_i \end{cases}$$

Typically, blocks are finalized in two to three epochs or approximately 12.8 to 19.2 minutes [81]. On average, a user transacts in the middle of an epoch. Thus, the last epoch only needs 66% of attestations, and thus, a transaction included there is finalized faster - averaging 14 minutes (16 slots from the first epoch + a full epoch, or 32 slots, + 66% of the last epoch, or 22 slots). Therefore, for use cases where an application on a third-party blockchain requires strong consistency on the Ethereum state, a safe buffer of around 10.8 to 19.2 minutes (averaging 14 minutes) is expected. Let us define this time as Δ_{final}. Typically, blocks are finalized in two to three epochs or approximately 12.8 to 19.2 minutes [81]. On average, a user transacts in the middle of an epoch. Thus, the last epoch only needs 66% of attestations, and thus, a transaction included there is finalized faster - averaging 14 minutes (16 slots from the first epoch + a full epoch, or 32 slots, + 66% of the last epoch, or 22 slots). Therefore, for use cases where an application on a third-party blockchain requires strong consistency on the Ethereum state, a safe buffer of around 10.8 to 19.2 minutes (averaging 14 minutes) is expected. Let us define the time necessary for the blockchain to finalize a block by Δ_{final}.

We observed that creating a SNARK proof on our hardware takes 4 minutes and 25 seconds. That is the current minimum latency, as one can generate a proof for the next block transition while generating the previous block proof. However, we defined the `SLOT_JUMPS` parameter on the worker to two epochs, or 64 slots (i.e., we batch transactions spanning two epochs). Since 64 slots × 12 seconds = 12.8 minutes, the latency introduced by the batching process supersedes the proving time. Therefore, $\Delta_{proof} = 12.8$ minutes. We chose this value to balance operational costs with liveness. In practice, we could further reduce latency to 6.4 minutes (32 slot jumps) and even lower to around 4.5 minutes, which is the time to generate a proof. Indeed, one can reduce latency by skipping fewer slots, at the peril of collecting transactions that will not be finalized and thus be included only in the next proof. It is worth noting that there is a trade-off between having smaller jumps and the time window for finalization: if a user submits a transaction in the second slot of the epoch (admitting the relayer will start creating a SNARK at the beginning of each epoch), the user will have to wait a full 31 slots until the prover starts with an input including that transaction. Conversely, the sooner a transaction is issued relative to being picked by the relayer, the more the user will have to wait for transaction finalization. Therefore, in minutes, $\Delta_{proof} < \Delta_{final} + \Delta_{tx_s}$ (minimum finalization time + a delta) $< \Delta_{snark} \lessapprox 25.6 + \epsilon$ (maximum finalization time + delays on relayer).

Empirically, for the case of Ethereum to Polygon direction, this translates into a delay of around 50 blocks relative to the source chain: we have verified a transaction included at block i at block $i + 50$. The target chain has a 150-180 block delay (will vary according to the specific destination chain): we can prove a fact at block header i in the source chain when around 150 blocks have been finalized in the target chain.

The Application Relayer constructs performs two reads from the blockchain state of the source and destination chains, creates two Merkle proofs, signs and broadcasts one transaction on the source chain, and signs and broadcasts three transactions on the destination chain, among other low-resource tasks. Since all of those happen concurrently with the SNARK Relayer, in a few seconds, Δ_{app} is statistically negligible. However, after the SNARK relayer has made the newest block execution state root available, the Application Relayer still needs to finalize the migration process according to our use case definition. The rest of the end-to-end process takes the time of one transaction confirmation. Since the SNARK latency will be the highest and parallel with the Application relayer and transaction finalization on different chains, we can approximate the overall latency: $\Delta_{total} \approx \Delta_{snark} + (|tx|_s \times \Delta^{Ethereum}_{\text{store}} + \Delta_{app} + |tx|_d \times \Delta^{Polygon}_{\text{store}}) \lessapprox \Delta_{snark} + (1 \times \Delta^{Ethereum}_{\text{store}} + \Delta_{app} + 3 \times \Delta^{Polygon}_{\text{store}}) \lessapprox 25.6$.

D. Performance Improvements

There several considerations we can make on performance improvements. First, one could take advantage of the light client update steps that do not have dependencies on other steps. For instance, we can parallelize the different checks done at `ValidUpdate` (line 1 of Algorithm 1) by distributing independent parts of proof generation to different machines (introduced in [32]). This would make the generation of SNARKs faster, reducing Δ_{snark}.

Secondly, let us focus on the fact that the previous analysis does not leverage the optimistic block header at the time of transaction submission (at time Δ_{tx_s}). Instead, we wait for finalization and instead use the finalized block header. We can improve this process by starting the SNARK generation at time Δ_{tx_s}. Furthermore, we can generate one SNARK *per step* in Algorithm 1 and then aggregate them by employing a recursive SNARK scheme [32], [82], [83]. Depending on whether all transactions chosen to be part of the current block at time Δ_{tx_s} (optimistic block header) get effectively finalized (finalized block header), we may have substantial latency reduction (namely Δ_{proof}). This is because we can now work with the optimistic block header instead of relying solely on the finalized block header. If the finalized block header is the same as the finalized block header right before Δ_{tx_d}, then our total latency will be the chosen finalization latency ($\Delta_{final} < \Delta_{final} + \Delta_{proof}$), as the proof is calculated concurrently with the execution of the finalization process.

Since we parallelize our SNARK generation process, we may have to recompute some SNARK phases (but not all). Thus, we hypothesize that the total latency would be $\Delta_{final} + m \times \Delta_{proof}$, where $0 < m \leq 1$ is the multiplier that represents

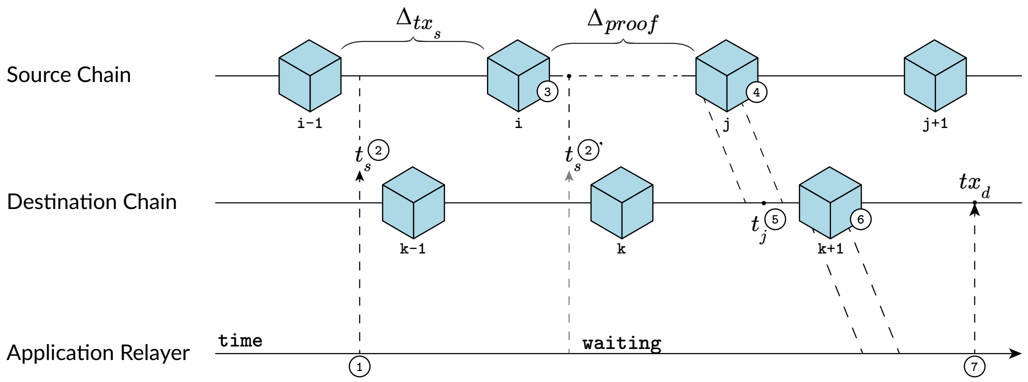


Fig. 8: Overall latency of a system built with Harmonia. It includes the latency to perform cross-chain transactions Δ_{tx_s} and Δ_{tx_d}, and the latency to generate a proof, Δ_{proof}.

the latency obtained by parallelizing the process. We leave the exploration of performance optimizations for future work.

E. Storage

A SNARK has a constant size: three points in the BN-254 curve. The total size is two 254-bit points, and another point is a bilinear pairing. On-chain, this takes 7×32 bytes = 224 bytes. Providing five updates per hour would yield a cost of 9.36 Mb per year (both on-chain and off-chain, ignoring database metadata). On the other hand, the storage cost for light clients to track the Ethereum chain is $\approx$ 25 kB per 27 hours: 24576 bytes for the 512 48-byte public keys in the sync committee, 96 bytes for the aggregate BLS signature (using a 381-bit curve), roughly 540 bytes per block header (e.g., parentHash is 32 bytes, a nonce is 8 bytes), and a Merkle tree branch. The storage occupied by the branch depends on the depth of the tree multiplied by a 32-byte hash. For a tree with a depth of 10, this would yield 320 bytes. In total, the cost for light clients to track the Ethereum chain would translate into ≈ 4.5 Mb per year. This amount of information would be unfeasible to store on-chain (for Ethereum, the price would be 50,000 USD per MB). However, two factors alleviate this problem. First, the light client sync committee works at the protocol level and thus does not pay gas to track the block headers. Secondly, storing 9.36 Mb of information in a Layer-2 blockchain is relatively cheap, as we will show next. Some alternatives, such as duplicating raw data on the third-party chain, would require significantly more storage. Therefore, the SNARKs we transact on-chain considerably reduce on-chain storage and, consequently, the price of operating an interoperability mechanism.

F. Hardware

The attractive properties of SNARKS come at a cost: proving correct even a simple computation is significantly more expensive in terms of more time and memory than is required for the computation itself. This overhead arises because the prover P "arithmetizes" the computation F, which involves expressing it as an arithmetic circuit C that is much larger than the description of F [84]. Thus, the hardware needed is on the high end of available commercial hardware. The costs of operating the Application Relayer are not significant because it runs on commodity hardware. We then focus on evaluating the hardware and transaction costs of the SNARK relayer. Depending on the cloud provider, the yearly price can go from $\approx$ 15K (Google's n2-standard-32) to 30K USD (Amazon's R5d.4xlarge). A bare-metal approach (Ryzen 7800 CPU, 128GB) would reduce the costs to less than 1K USD per year but requires hardware maintenance.

G. Transaction Fees and Costs

We analyze the transaction costs of the Light Client Verifier contract (Beacon Light Client), the Application Proof Verifier, and the cross-chain logic contracts regarding gas. We consider the costs of deploying the Light Client Verifier (1,399,127 gas) and performing one light client update (279,963 gas). We calculate the yearly costs in USD for issuing three light client updates per hour (1 every 20 minutes). This gives an overview of the transaction costs to operate the light client in several EVM-based chains (see Table I):

Regarding our cross-chain logic contracts, our integration adds 7,225 gas (around 40 cents), including one extra call to DendrETH to retrieve the execution state root. The values to obtain a storage variable are around 2,400 gas, and to set the variable, they are 5,400 gas. Deploying the Relay, Proxy, and simple storage contracts cost 1.6M, 3M, and 140K gas, respectively. Performing the migration process costs around 950K gas, costing around 0.2 MATIC (17 cents) in Polygon.

In the context of blockchains with a WASM runtime (e.g., Polkadot, Cosmos, Elrond, NEAR, EOS, Fantom), the verification costs of a SNARK is around 250K gas and over four times that for the direct verification (i.e., verifying consensus rules in the smart contract). Table II shows the gas costs for deployment and updates of our different implementations of the SNARK verifier.

H. Considerations on Throughput

Throughput depends on the specifics of the use case (i.e., cross-chain logic rules), the latency of each transaction, upper-bounded by the throughput of the underlying chain, and the

	Price/Gas	**Rate USD/Native Token**	**Light Client Verifier Deployment (USD)**	**Light Client Verifier Update (USD)**	**Cost/Year (USD)**
Ethereum	29 Gwei	2010	81.61	16.33	429,152
Polygon	200 Gwei	0.85	0.24	0.05	1,314
Avalanche	26 nAVAX	14.79	0.54	0.11	2,891
EVMOS	30 nEVMOS	0.09	0.004	0.001	26.28
Optimism	0.001 Gwei	1.37	≈ 0	≈ 0	0.01
Arbitrum	0.1 Gwei	1.22	0.0002	≈ 0	0.9

TABLE I: Light Client Verifier deployment and Update costs

	Deployment	**Initialize**	**Update**
NIM-WASM Light Client	1,308,702	2,991,395	11,706,455
SNARK Verifier using nim-bncurve	1,302,849	447,436	1,812,337
SNARK Verifier using constantine	1,378,889	391,408	871,846

TABLE II: Deployment, Initialization, and Update Gas Costs

specific block generation rates of the source (s) and target (t) chains. This is due to the existence of a delay δ from transaction broadcast that affects the non-deterministic block generation rate τ, which will affect the liveness of the system. The ratio between block production in the source and target chains, $\tau_s^d = \frac{\tau_d}{\tau_s}$, sets the speed at which one can prove facts on the target chain.

For use cases that require an interdependency of transactions across chains enforced by on-chain verifier contracts (such as our PoC), throughput will depend on the latency Δ_{proof} for the proof verification. In our case, we execute four transactions in around 25 minutes, translating into a throughput of 1 finalized, irreversible transaction every 6.25 minutes. The bigger the batch of transactions contained in a single SNARK proof (which contains attestations for block headers execution roots), the higher the throughput. We did not optimize our PoC for throughput, and we leave the study of the performance of more complex cross-chain use cases for future work.

I. Reproducibility

We provide tools for researchers to set up our project and reproduce our empirical results easily. First, we make available our codebase and results under a permissive open source license[15]. We provide tests and Docker containers following recommended engineering practices [85], [86]. Tests are available for the on-chain smart contracts, direct implementation, and Circom circuits. To facilitate our testing infrastructure, we maintain an archive of light client updates for each sync committee period since Altair, as produced by a fully-synced Nimbus node, available for the Ethereum mainnet and Prater [16]. For Prater/Goerli, the updates start on checkpoint 5601 823. Pre-generated proofs have been available since that checkpoint. Furthermore, to ease setting up the environment, we leverage Nix, a package management and system configuration tool that helps us showcase a reproducible, declarative, and reliable system. The deployment addresses of our contracts are available in our project's README.md file.

VI. DISCUSSION AND QUALITATIVE ASSESSMENT

In this section, we present the discussion and qualitative evaluation of our solution.

A. Safety

In this section, we analyze long-range light client sync attacks in the context of the safety of our system.

Altair 1.0 is vulnerable to bribing. Although individual action by a small set of validators would be insufficient for a large-scale attack, the industry has seen centralized platforms for influencing validator behavior. An example is the Flashbots platform, which provides auction-based coordination to extract value from block reorganization [63]. Therefore, cooperation to exploit Ethereum's light client protocol could emerge sooner or later as an obvious first venue for conducting cross-chain attacks [12].

To harden the system's safety against these attacks, DendrETH is a necessary improvement to the state of the art, currently being standardized and implemented [87]. However, this security model might not work for smaller chains with much smaller total value locked (meaning bribes and colluding prices would be cheaper). Given that a high volume of traffic of a popular interoperability solution is non-EVM (order of tens of billions USD), this indicates the need to study the crypto-economic attacks possible to do for the light clients of those infrastructures. An example is the elevated traffic between layer two solutions, namely Arbitrum and Polygon, where the users want to transact between chains and not wait the large waiting queue periods to withdraw their funds from the L2 (typically using an optimistically verified method). Although this is not easily fixable, one could instead closely monitor misbehaving, forks, and slashing on these smaller chains using cross-chain models, and use circuit breakers for cross-chain logic if some suspicious behavior is detected.

Still, regarding safety, let us discuss long-range attacks on Ethereum light clients. In Proof of Stake Ethereum, nodes must

---

[15] available at Github, https://github.com/metacraft-labs/DendrETH

[16] see light client updates, https://tinyurl.com/yampz8re.

verify block headers, account states, and balances throughout the blockchain history or consider the risk of long-range attacks [43], [88]. The attack involves an adversary taking over the blockchain by creating an alternative (forked) chain starting from a point deep in the history of the legitimate chain. In PoS systems, the ability to create blocks is proportional to the amount of stake (tokens) one holds. However, in the past, it is possible that a large amount of stake was held in addresses that now hold little to no stake. If the private keys of these old, 'empty' addresses are obtained (perhaps they were discarded, sold, or are no longer secured because the stake was moved), an attacker could theoretically create a new chain starting from when those addresses had a significant stake. Given enough time, this alternative chain could grow longer than the original chain. In blockchain protocols, the longest chain is often considered the 'correct' one, so this could enable the attacker to overwrite the blockchain's history.

A long-range attack could trick light clients as they inherently trust the validity of the blockchain headers they receive and do not fully validate the entire blockchain. If an attacker successfully executes a long-range attack and creates an alternate, longer blockchain, they could send the light clients the headers from this fraudulent chain, and verify false facts on a destination chain, conducting different types of attacks on cross-chain logic, because the real blockchain history is indistinguishable from the forged one. The counter measure employed by Ethereum is to require all clients always to start syncing from a trusted recent checkpoint, which guarantees that the maximum number of exited validators will not be sufficient for carrying out an attack. This notion is known as weak subjectivity [89].

We now perform a detailed analysis of weak subjectivity from the perspective of light clients [90] to calculate the risk of long-range attacks. We consider the number of active validators as of 14 July 2023, $664,205$. The light client software may be pre-configured with a trusted bootstrap state, and it may rely on SNARK proofs (as developed by the DendrETH project) to perform one-shot syncing over a long range. We focus on two evaluation experiments: 1) what is the ratio of validators that can be malicious without the light client committee being corrupted? 2) what is the threshold of exiting nodes such that a malicious supermajority could be formed and can carry out a long-range sync attack? and 3) what is the risk profile for our validator pool, with security parameter $= \frac{2}{3}$?

1) Experiment 1 – Malicious Validator Ratio: We define our security parameter λ_s as the percentage of the sync committee that signs a block in order for it to be considered part of the canonical chain. Figure 9b shows the probability of corruption of the light sync protocol P_c (i.e., forming a malicious sync committee $\mathcal{M}$ supermajority) as a function of the percentage of malicious nodes with λ_s fixed at $\frac{2}{3}$ (default). We run an approximation to the binomial cumulative distribution function (cdf) to calculate the safe percentage of malicious nodes, expressed in the following inequality:

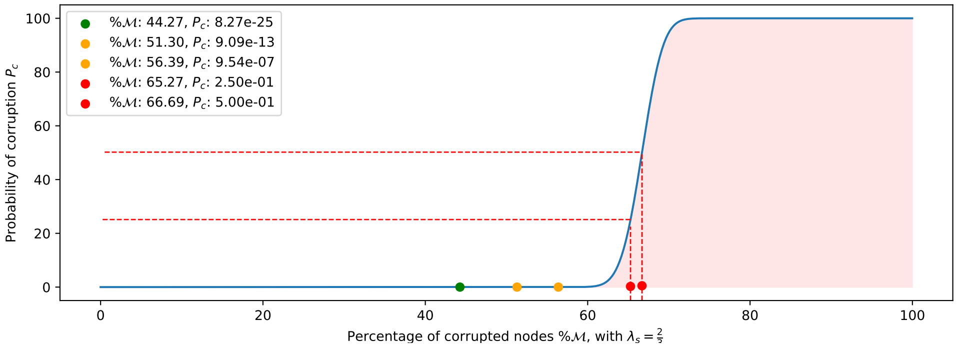
$$\sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{(n-k)} \leq e$$

Where n is the total number of selected validators (total 512, the light client committee size), from the total set of Ethereum validators, k is the number of honest validators, e is the error, and p is the probability we want to calculate. For the probability of corruption to be $2^{-80}, 2^{-40}, 2^{-20}, 25\%$ and 50%, the percentage of malicious validators need to be $44.3, 51.3, 56.4, 65.3$, and 66.68, respectively. For example, to calculate a probability of corruption of 50%, the inputs are $n = 512$, $k = 256$, and $e = 0.5$. Note that for values until around 60%, the probability of corruption is practically zero but increases exponentially. Figure 9a represents the variation of P_c as a function of λ_s, starting at $\frac{2}{3}$. We can observe that for $\lambda_s = 70, 80, 90$, $\mathcal{M} = 47.7, 58.7$, and 71.4 respectively.

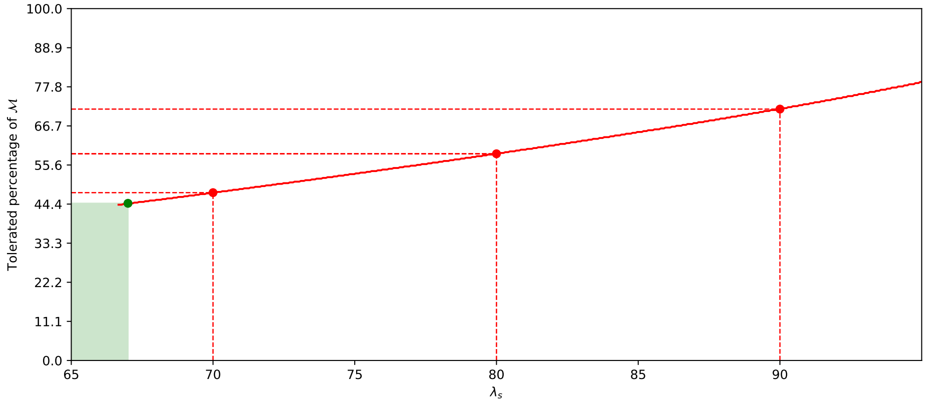
2) Experiment 2 – Churn Rate Safety Thresholds: Let us fix $\%\mathcal{M} = 44.3$, for which $P_c = 2^{-80}$. We assume the worst-case scenario: every validator exiting the validator set will immediately turn malicious. As discussed previously, long-range attacks target Ethereum clients who have not synced with the blockchain for a while. Since they have an outdated view of the chain, they cannot know which validators have exited (or got slashed). The malicious actor only needs to create a single light client update with a super-majority of the signing keys that correspond to this old state of the light client in order to lead it into an alternative history where each next sync committee is also under the control of the attacker.

Considering the churn limit and the percentage of malicious validators needed to compromise a sync committee (Figure 9), we can calculate exactly how long it would take for those many validators to exit. First, we define the churn limit $\overline{c} : \mathbb{N} \rightarrow \mathbb{R}$, where $\overline{c}(|\mathcal{V}|) = \frac{\max(4, |\mathcal{V}|)}{2^{16}}$, where 4 is the minimum number of validators that can exit an epoch, 2^{16} is the churn limit quotient, and $|\mathcal{V}|$ are the number of active validators. The number of epochs required for a validator to exit the system ee is given by $ee = \frac{|\mathcal{V}|}{\overline{c}(|\mathcal{V}|)} \times exit$, where $exit$ is the fraction of validators exiting the system. Now we vary $|\mathcal{V}|$ and $exit$ (Figure 10). We conclude that for an exit percentage of 44.3% (has a probability of 2^{-80} for that set to include a supermajority of malicious nodes), validators would need to wait 129 days. For higher percentages (e.g., 51.2%, 56.3%), validators would have to wait 149 and 164 days, respectively. Thus, we can conclude that if a light client seeks a sync committee honest super-majority of at least $\frac{2}{3}$, it is safe to sync at least once every 129 days. Another way to increase the robustness of light clients is to increase the ratio r of honest validators to sign light client updates (see Figure 15 in Appendix E).

3) Experiment 3 – Fine-Grain Risk Assessment: In this assessment, we use the hypergeometric cumulative distribution function to compute the probability of forming $\mathcal{M}$ given a fixed λ_s. Although the results are similar to the binomial function, the hypergeometric cdf ($F(k; N, M, n)$) considers



(a) Tolerated percentage of $\mathcal{M}$ as a function of λ_s



(b) Probability of malicious majority relative to share of malicious validators

Fig. 9: Sync committee corruption simulations

that once an exited validator is selected for a committee, it exits the pool of potential validators to be selected. The formula we use is the following:

$$F(k; N, M, n) = \sum_{x=0}^{k} \text{PMF}(i; N, M, n)$$

$$\text{PMF}(i; N, M, n) = \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}}$$

Where $i \in [\frac{2}{3}, 1]$ (goes from $\frac{2}{3}$ of 512 to 512, N is the total number of validators, rounding it up for the closest multiple of the churn limit quotient ($N = 720896$), M represents the population that has been churned, and $n = 512$. The risk levels (exponent of the probability of $\mathcal{M}$ (powers) occurring in $\mathcal{V}$) are the same considered in previous experiments (probability of corruption = $2^{-80}, 2^{-40}, 2^{-20}$). We calculate the risk of a malicious majority occurring in a validator set (powers of the corruption probability) as time passes and validators churn in Figure 11. We notice slight discrepancies relative to Experiment 1 because the hypergeometric cdf is more precise than the approximation with the binomial cdf. For $\mathcal{M} = 44.3\%$, we expect to wait approximately 125 days, a negligible risk operation.

B. Liveness

The liveness of our system is tied to the liveness of the light client sync committee and the liveness of off-chain parties. For the liveness of off-chain parties, techniques like crash-recovery for blockchain clients can be deployed [91], as well as having multiple instances deployed, mitigating the probability of unavailability.

We now explore on-chain liveness: let us consider the case where the light client receives a valid update containing a $finality\_header$ with at least two-thirds of the sync committee participating. However, there might be cases where part of the sync committee is unavailable (crash or attack). If the light client sees (no valid updates via the method for a one-sync committee period), it simply accepts the speculative header with the most signatures as finalized. This allows the light client to be live even during periods of extended non-finality, though at the cost of network latency of a period. This downtime period is not sufficiently long for long-range attacks.

Cross-chain protocols are conventionally structured to manage such situations efficiently by initiating retries following the resumption of services. These anomalies are deemed harmless up to a liveness parameter. Effectively, upon crashing, the

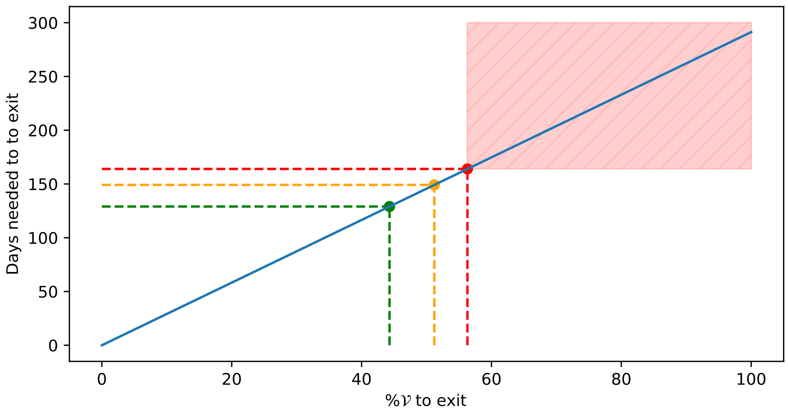


Fig. 10: Days needed for a fraction r of validators that form a malicious supermajority to exit Ethereum

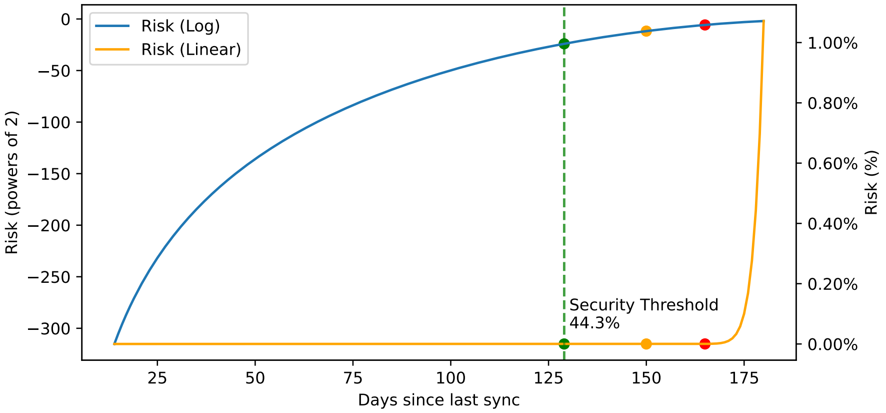


Fig. 11: Cumulative probability of $\mathcal{M}$ (powers) occurring in $\mathcal{V}$ as time passes and validators churn.

relayer keeps track of the last update and submits all late updates to the chain when it is back online or when the chain resumes its operations, similarly to the crash fault recovery protocol of blockchain gateways [7], [91]. Effectively, this has happened a few times on our testnet deployments, as we can see in the subsequent transactions after this one [17]. The relayer provided all light client update transactions missing quickly after the incident.

C. Accountability and Auditability

Light client initialization (via `INIT`) is done either on the genesis or pre-agreed blocks. Assuming the genesis block or the pre-agreed block is valid (which can be done by re-executing the whole chain or social consensus), incoming light client updates are valid. One can aggregate light client updates in a data structure that promotes traceability and auditability from period i up to period j: $\texttt{LightClientStore}_j \rightarrow (\mathcal{L}_{\mathcal{S}_{[i,j]}})$. One can run `LCU` from state i up to j and verify that the outputs are always valid. Zero-knowledge SNARKs (zk-SNARKS) enable a prover to convince a verifier that a given statement is true without revealing any additional information (unlike SNARKS, which do not have the zero-knowledge property). Using zkSNARKs does not come with advantages to our use case because it creates trade-offs between the recorded evidence on-chain that can be used to slash any corrupted members of the sync committee (see the specification of DendrETH, Figure 5).

D. Censorship Resistance

Attackers could attempt to block valid light client updates passing to the deployed chain, violating censorship resistance, by performing a denial of service on the off-chain relayers. A group of validators could attempt a denial of service, which would depend on the decentralization and security of the attacked network. Although a decentralized network of relayers would alleviate the likelihood of traditional denial-of-service attacks, an attacker could explore an alternative route. In particular, a denial of service of the whole blockchain could try to be performed. An attacker could buy the entire block space for the duration of an attack, which is around 5

[17] see PolygonScan, Mumbai network, https://tinyurl.com/d9mtusny

ETH per block in July 2023[18]. To control the inclusion of blocks for a day, an attacker must spend at least 580 ETH, approximately 1 million USD. This would effectively censor 120 light client updates. Since the block space market is very dynamic and unpredictable [92], it is extremely unlikely that the adversary can select the gas fees to be higher than in every other transaction and the attack is economically viable. The worst-case scenario brings difficulties for an attacker. For example, the peak of gas price was ≈ 710 Gwei in July 2020, making a single Eth transfer cost 0.015 Eth. Since the gas limit in a block is 30 million [93], buying a single block could cost up to 21.3 Eth (42K USD). Since 7,200 blocks are created daily, this could account for $\approx 304,704,000$ USD to censor the entire network daily.

Note that this value could be considerably lower for weaker security blockchains. In Polygon, the daily block rewards are around 40K MATIC, around $26,800 USD. Thus, running a DoS for the light client where the source chain would be Polygon would probably cost substantially less (low fees are a feature of Layer 2 technologies). However, the specifics of an attack would have to be further investigated, as market forces could influence the gas fees consumed by a denial-of-service attack in unexpected ways [94].

E. Upgradeability, Flexibility, and Extensability

A successful dApp has a certain degree of adaptability (capacity to fix bugs and add features). We can account for this need for adaptability and hard forks on the Altair protocol by deploying the verifier contracts behind a proxy contract. The proxy should only be updated upon a quorum of trusted validators. However, updating contracts comes as a double-edged sword, as the upgradeability of contracts is an attack vector that has led to the loss of hundreds of millions USD recently [95]. In case the circuits are updated, another trusted ceremony must be performed. One can explore different proof systems to circumvent the need for a trusted ceremony (such as STARKs [96]). However, STARKs produces larger proofs, which means larger gas consumption by blockchains, which translates to higher fee costs. STARKs also require more time to generate and verify the proof.

Harmonia should provide The ability for use cases to adapt to such protocol changes while minimizing disruptions and risks to the cross-chain use case. For this, it maintains a well-defined and documented schema that cross-chain use cases can consume (API abstraction) and adopts good practices used in the industry (providing the necessary Flexibility, $\mathcal{G}_5$). With additional engineering effort, we can add more chains supporting the verification of the source chain's state by implementing a custom SNARK verifier (satisfying Extensibility, $\mathcal{G}_5$). This satisfies the direction Ethereum $\longrightarrow$ other chains. To provide the direction chain $\longrightarrow$ Ethereum, one would need to implement the light client protocol circuits of the origin chain for them to be consumed by Ethereum. Working with both directions would allow us to implement a trustless bi-directional bridge, which we leave for future work.

F. Security Analyisis

In this section, we prove the security of the Harmonia system by proving the security and liveness of its components. Proving these properties provides a contribution to solving the safety and liveness challenges presented ($\mathcal{C}_1, \mathcal{C}_4$). For this, we leverage the concept of composability. Composability in the context of system security refers to the ability to design and validate a system by combining independently validated components or subsystems. Proving that a system composed of multiple subsystems is secure involves demonstrating that the security properties of the individual subsystems collectively ensure the security of the entire system under composition.

Let us recall the desired goals: Safety ($\mathcal{G}_1$), Decentralization ($\mathcal{G}_2$), Finality ($\mathcal{G}_3$), Liveness ($\mathcal{G}_4$), Extensibility ($\mathcal{G}_5$), Flexibility ($\mathcal{G}_6$), Off-chain safety ($\mathcal{G}_7$), and Censorship-Resistance ($\mathcal{G}_8$). We prove goals ($\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3, \mathcal{G}_4$, and $\mathcal{G}_7$). Goals $\mathcal{G}_5$ and $\mathcal{G}_6$ are achieved, as described in Section VI-E. We consider a probabilistic polynomial time adversary $\mathcal{A}$. An important consideration for the security proofs is that we consider a partially synchronous environment, where the global stabilization time is $\approx$ 129 days (we deem this model appropriate; see Experiment 2 from Section VI-A).

Proving these properties hold for Harmonia is proving they hold for each component, although we might only have to prove a subset of all the properties, depending on the component. Let us recall the components: source chain ($\mathcal{C}_s$), destination chain ($\mathcal{C}_d$), light client verifier contract ($\mathcal{SC}_{lc}$), application verifier contract ($\mathcal{SC}_{app}$), cross-chain logic contract on the source chain ($\mathcal{SC}_s$), cross-chain logic contract on the destination chain ($\mathcal{SC}_d$), SNARK relayer ($\mathcal{R}_{snark}$), and application relayer ($\mathcal{R}_{app}$). The security models for each component have been presented throughout the paper (mostly in Section III). The interactions between components have also been presented (Figures 1 and 2).

The source and target chains are both blockchains, so the proof is the same:

Theorem 1. *$\mathcal{C}_s$ and $\mathcal{C}_d$ provide safety and liveness.*

PROOF: By assumption, 1) both chains have n nodes, and malicious nodes f are upper bounded $f < \frac{n-1}{3}$.; 2) the properties of consistency, chain quality, and liveness. Consequently, both chains are secure under our definition.

Corollary 1. *Smart contracts $\mathcal{SC}_{lc}$, $\mathcal{SC}_{app}$, $\mathcal{SC}_s$, $\mathcal{SC}_d$ are correctly executed and live*[19].

PROOF: The correct execution of smart contracts is tied to the security and liveness of the underlying infrastructure. For $\mathcal{A}$ to force incorrect execution of any of these contracts, they would have to break the security of the underlying chains. Following Theorem 1, this is a contradiction.

---

[18] see Etherscan, https://etherscan.io/chart/blocks

[19] Nonetheless, correct execution does not assure that the contracts correctly implement the specification. In practice, there may be implementation bugs.

Theorem 2. *SNARK relayer $\mathcal{R}_{snark}$ is safe, live, decentralized, and provides (weak) censorship resistance.*

`PROOF`: Let us first prove the safety of the relayer. The safety of the relayer is tied to the correctness of the SNARK proof system generating the SNARKS. In particular, Groth16 is known to be computationally secure under certain hardness assumptions: the knowledge of exponent assumption and bilinear strong Diffie-Hellman [47]. Generating invalid proofs for Groth16 involves solving these problems, which are known to be computationally hard. Therefore, the relayer cannot generate invalid proofs, and safety is assured. The SNARK relayer liveness is tied to the liveness of the ALC (see Section VI-B) and off-chain liveness. Off-chain liveness is assured by line 3 of Protocol 4, and it can be hardened by employing crash-recovery capabilities and consuming from multiple infrastructure providers [8]. Since 1) the light sync committee (via DendrETH) is live and can support downtime up to $\approx$ 129 days, 2) the relayer employs crash-recovery techniques, we deem our SNARK relayer live. The relayer is decentralized because anyone can run a SNARK prover, provided access to the necessary hardware, and submit proofs on chain (all the code is open-source, the SNARK verifier is not permissioned), see Protocol 4. Finally, the censorship-resistance can be decomposed into on-chain and off-chain. For on-chain censorship resistance, an attacker must spend at least 8,333 USD to censor a single light client update (see Section VI-D). To censor relayers, $\mathcal{A}$ needs to bribe them all not to publish light client updates. Since relayers will be incentivized by rewards and reputation and backed by individuals and enterprises, an attacker is unlikely to censor them all for an extended period (weak censorship resistance).

Theorem 3. *The Application relayer $\mathcal{R}_{app}$ is safe, live, decentralized, and provides (weak) censorship resistance.*

`PROOF`: Let us first prove the safety of the relayer. The safety of the relayer is tied to the correctness of Merkle proofs. In particular, the correctness of Merkle proofs depends on the element used as the source of truth: the block header root. Since the sync committee validates this element, the security of these proofs is tied to the security of the light client committee. Generating invalid proofs for DendrETH requires bribing a supermajority of the light client sync committee, costing $\approx 20M$ USD for a sync period (Section III-B). The liveness is tied to the liveness of the off-chain infrastructure. Line 2 of Protocol 5 guarantees that the application relayer keeps updating a cross-chain state. Similarly to the SNARK relayer, liveness can be hardened by employing crash-recovery capabilities and consuming from multiple infrastructure providers [8]. The relayer is decentralized because anyone can generate Merkle proofs provided access to the source blockchain. and submit proofs on chain (all the code is open-source, the application verifier is not permissioned), see Protocol 5. Finally, the censorship-resistance can be decomposed into on-chain and off-chain. For on-chain censorship resistance, an attacker must spend 1M USD to censor all transactions in a block (but it could be much more; see Section VI-D). To censor relayers, $\mathcal{A}$ needs to bribe them all not to publish Merkle proofs. Since relayers will be incentivized by rewards and reputation and backed by individuals and enterprises, an attacker is unlikely to censor them all for an extended period (weak censorship resistance).

Let us now compose the previous theorems and provide a proof for Harmonia:

Theorem 4. *Harmonia is a decentralized system providing safety, liveness, and censorship-resistance.*

`PROOF`: We prove Harmonia satisfies the defined set of goals. One needs to ensure that:

1) $\mathcal{C}_s$ and $\mathcal{C}_d$ are live and secure and support smart contracts (according to Section II-A).
2) Smart contracts $\mathcal{SC}_{lc}$, $\mathcal{SC}_{app}$, $\mathcal{SC}_s$, $\mathcal{SC}_d$ are correctly executed and live
3) there is at least a safe and live SNARK Relayer $\mathcal{R}_{snark}$
4) there is at least a safe and live Application Relayer $\mathcal{R}_{app}$
5) the source chain has a sound, succinct, and live light client protocol (according to Section II-D).

Point 1 follows from Theorem 1. By definition, chains support smart contracts. Point 2 follows from Corollary 1. Point 3 follows from Theorem 2. Point 4 follows from Theorem 3. We need to prove that the source chain has a sound-light client protocol. Harmonia uses DendrETH as its light client protocol. DendrETH's succinctness is trivial because the sync committee provides a small subset of information from the chain (validated block headers). The liveness of DendrETH is guaranteed as long as a supermajority of the committee is online (economic analysis in Section VI-B). Downtimes are supported for up to 129 days. Finally, the protocol is sound as 1) a supermajority of malicious nodes belonging to a sync committee has a negligible probability (see our probabilistic analysis in Section VI-A); 2) DendrETH punishes malicious validators who attest to invalid information □.

G. Trusted Ceremony and Initialization

Systems built with Harmonia will have to conduct a trusted ceremony with industry partners to ensure the robustness of the process. Some operational challenges exist: coordinating the different parties and eliminating the toxic residues. Typically, at least two key pairs need to be generated: one for the prover and one for the verifier. However, the input parameters of this generator algorithm must be secret, meaning that they must be hidden for both the prover and verifier; otherwise, the scheme's security can be broken. To ensure the operators do not unfaithfully learn the private parameters, one can perform the setup process publicly by a set of mutually untrusted parties. Different technologies, such as multi-party computation, could eventually be used. In this setup, n parties engage in a protocol such that each one generates a partial key. Individuals cannot learn the key without input from other $n-1$ parties. Therefore, the security of this setup relies on at least one honest participant eliminating their share of the verification key. For an example, see the ZCash blockchain key generation ceremony [97], [98].

Furthermore, the ceremony should have a plan that is transparent and enforces the publication of all steps, software, and hardware specifications. Transparent planning should also incentivize auditability and accountability (e.g., penalize parties that leave the ceremony midway). Some other good practices are promoting public participation of a large and diverse group of participants, as Ethereum did with the KZG Summoning Ceremony[20], ensuring the privacy of the participants, and the usage of secure communication channels. After this one-time ceremony, an arbitrary number of proofs can be generated and verified for the current circuits.

H. Post-Quantum Considerations

Groth16 relies on discrete logarithms being difficult to compute, so the particular proving scheme we instantiated Harmonia with is not post-quantum secure. A quantum computer can very efficiently factor integers using Shor's algorithm, which can be used to break elliptic curve cryptography [99]. Current quantum computers are still incapable of breaking elliptic curve cryptography, although recent developments pose a not-too-distant timeline.

Preliminary research into post-quantum zero-knowledge proofs shows that zk-STARKs (Zero-Knowledge Scalable Transparent Arguments of Knowledge) [100] and lattice-based SNARK constructions [101], [102] are quantum resistant. Zk-STARKs use hash functions and error-correcting codes instead of elliptic curve pairings and do not need a trusted setup. However, zk-STARKs are generally more computationally intensive and generate larger proof sizes compared to zk-SNARKs, which incurs an extra cost to on-chain proof verification[21]. The performance trade-off is important, considering the development timeline of quantum cryptography.

I. Incentivization

The system has to use incentives to be able to keep the different actors performing their functions. First, relayers have to be incentivized to operate the key infrastructure. In some cases, relayers can act *pro bono*, being sponsored by foundations responsible for the maintenance of the ecosystems. In other cases, infrastructure companies like Blockdaemon or individuals can run these relayers for a fee that covers operational costs. For this scheme to be dependable, the cost of computing SNARKs off-chain must surpass the cost of executing the light client protocol on-chain, even at the cost of running expensive infrastructure. Executing the equivalent algorithm on-chain is not feasible due to the lack of EVM precompiles for verifying BLS signatures. Such primitives have been proposed [104], but this proposal is stagnant.

A prerequisite of any incentive scheme is unstealability, i.e., the guarantee that one's work is not stolen, which is encapsulated by the free rider problem [105]. The free rider problem stems where MEV bots could sniff proofs [106] (the input to the verifier smart contract) encoded in smart contract calls, present in transactions awaiting block inclusion in the transaction memory pool. We propose three solutions. The first and most centralized solution is creating a privileged role in the smart contract that can be lost if a relayer does not submit the proofs promptly (in other words, if the operator goes out of business, any user can step in to become the operator). The second solution consists of relayers transacting in a round-robin way, creating a more inclusive network. Finally, one could use a commit-and-reveal scheme on the verifier contract that receives the sender's identity as input. However, this solution involves an extra transaction (possibly only 32 bytes, a hash), carrying more on-chain costs.

Different business models based on fees can exist. For example, the light client may allow applications to consume the validated information for a fee, building a treasury and paying the relayers. A detailed crypto economic analysis is out of the scope of this paper, but one could decentralize the relayer network with the incentivization mechanism behind projects such as some state-of-the-art interoperability DeFi protocols in the industry [107]. For instance, a token can be introduced to allow voting on the system's parameters (including the choice of the relayer) using a decentralized autonomous organization. Currently, our system is entirely decentralized and permissionless: any Dendr Harmonia ETH relayer can freely join the system to relay block headers and claim rewards.

J. Extending light client security to the whole validator set of Ethereum

Our system assumes that the price of performing an attack on Ethereum's light client surpasses the light client's economic security. However, we can increase the economic security of our system by verifying Ethereum's finality algorithm [108] (Casper Finality Gadget) rather than the correctness of the light client protocol execution. Like DendrETH, this scheme would allow us to prove facts happening on Ethereum using a SNARK. However, now the threat model shifts from a subset of the Ethereum validators to the entire validator set, greatly improving the security of cross-chain operations.

On a high level, this update could be realized by implementing a circuit that verifies Casper: 1) the prover obtains a list of aggregated attestations by monitoring the gossip network and the history of proposed blocks, 2) the attestations are grouped according to their sub-committee and their signing root, 3) a circuit verifies that the signatures in the collected aggregated attestations match the aggregated public keys of the validators participating in the particular sub-committee, 4) another circuit aggregates proofs generated by the first circuit to obtain tuples in the form (source, target, attesting balance), and 5) a final circuit implements the Casper finality conditions, as defined by the latest version of the Ethereum consensus specs. This boils down to demonstrating that two consecutive epochs have attesting balance above two-thirds of the total active balance of

---

[20] see ceremony page, https://ceremony.ethereum.org.

[21] for the interested reader, the following paper provides a walkthrough of a STARK proof [103].

the beacon chain. We published a more detailed specification here [109].

VII. Related Work

In this section, we present the related work. We highlight a recent survey on the security and privacy of blockchain interoperability mechanisms, which this work is closely related to [14].

A. Blockchain interoperability

The interoperability design space is increasingly diverse [1], [2], [4], [14], [110]–[112]. To better navigate the various classification frameworks available, one can aggregate the interoperability solutions by *interoperability mode*: data transfers, asset transfers, or asset exchanges. Data transfers allow for arbitrary cross-chain use cases [113] and typically use light clients to verify facts on the source chains. Asset transfers follow the burn-mint or lock-unlock model and are implemented by cross-chain bridges [114] and blockchain gateways [7]–[9]. Asset exchanges use time locks and hash locks to perform a set of locally verified transactions that implement cross-chain asset transfers. Popular implementations include HTLC-based systems [115], [116], which can be used as underlying primitive adaptor signatures, hashes, or other mechanisms. In our paper, the PoC we showcased performed data transfers based on a trustless light client, showcasing how one can build oracles using SNARK technology. Asset transfers are also implementable with Harmonia, where a relayer constructs a proof for a lock/burn mechanism in Ethereum and feeds that proof into a target chain. For a recent overview of the interoperability area, see [1]. For comprehensive surveys, see [4], [14].

B. Light Client Protocols

There has been extensive work on light client protocols for different types of blockchains. In [117], the authors propose smart-contract-based light clients. In [118], the authors propose fraud proofs to enhance light clients. Other optimizations have been proposed, e.g., using probabilistic block sampling [22]. For a comprehensive theoretical overview of this area, refer to [43], [119].

In the industry, there are a few popular light client protocols. Cosmos uses the Tendermint BFT Proof of Stake consensus. The InterBlockchain Communication (IBC) protocol is a protocol providing interoperability for many Cosmos-based chains. It works with an underlying set of light clients [120]. It allows Cosmos zones, which are sovereign application-specific blockchains to send messages via an IBC channel. A receiver zone verifies messages in a third-party zone by verifying Merkle proofs rooted in validated block headers provided by a light client of the sender zone [121]. The BTC Relay is a Bitcoin light client on Ethereum. It is used to store Bitcoin block headers that can be used as a source of truth for proving facts (transactions, state) about Bitcoin, on Ethereum. It is now deprecated. Different variations exist, including the BTC Relay for Polkadot [122]. A use case is enabling the creation of vaults containing cryptocurrency-backed assets (enabling asset transfers from Bitcoin to Polkadot) [122]. For Byzantine fault tolerant-based consensus (e.g., Elastico, Omniledger, Algorand) [123] or crash-fault tolerant consensus (e.g., Hyperledger Fabric [124]), a light client needs to verify validator signatures and keep track of validator rotation. For private blockchains (e.g., Hyperledger Fabric with privacy settings on), the techniques rely on a trusted quorum that operates the private blockchain, blockchain views, and the usage of a decentralized public bulletin [27], [125], [126].

C. Comparison with other interoperability approaches

DendrETH can be classified as a natively verified state-of-the-art system [127], where the "destination chain independently verifies that the received state is valid and final according to the source network's state transition and consensus rule." [1]. Compared to other approaches, this provides a higher level of security, since the security model is based on the sound cryptography of SNARK technology and not on external systems. Locally verified approaches often require a trusted off-chain coordinator (which has been consistently exploited [12], [16], [128], incurring in a loss of around $2 billion), suffer from capital inefficiency (because assets need to be locked), and are vulnerable to sore loser attacks [129]. Optimistically verified systems imply a large latency for moving capital in the order of several days. Externally verified systems, the most centralized ones, are especially vulnerable to collusion and standard cybersecurity attacks targeting bridge operators, which are often centralized [130]. Our system provides a new security model based on the soundness of the underlying SNARK protocol and crypto-economics, with minimal assumptions (liveness of the relayer network).

D. SNARK-based cross-chain bridges

A primer was proposed by Westerkamp and Eberhardt [131], where the authors designed a SNARK-based verifier for Bitcoin, that yields validated block headers that can be consumed in Ethereum. However, this work focused solely on Bitcoin, and relied on a network of known peers. Recently, some projects have appeared to validate the consensus of more complex blockchains such as Ethereum. The following validate Ethereum's light client protocol[32], [132], [133], but these currently only work for EVM-based chains (except for zkBridge). While some of these have lower latency on the proof generation time, there is no practical benefit in reducing the latency beyond an Ethereum transaction's finalization time for a client application to consume a finalized root. We have not found code for relayers or on-chain verifier contracts for these projects. Furthermore, we have not found details on cross-chain applications built on top of these solutions, making it difficult for us to assess and compare these works systematically. Some circuit implementations prove other light client sync protocols (e.g., Bitcoin [134], ZCash [135]).

Indirectly related work includes solutions like Webb [136], that utilize zero-knowledge proofs to attest the validity of proofs used in the interoperability system (e.g., Merkle proofs on Ethereum or a hash of an unspent transaction output in Bitcoin). While a promising venue for private cross-chain interoperability, zero-knowledge proofs are not used to validate the correctness of light client updates. If used naively, performance-wise, it is cheaper to validate Merkle proofs rooted on an SNARK-verified block header than to validate Merkle proofs with an SNARK. Recent work proposes an efficient commitment scheme to be used in creating accountable light client systems [137].

E. Rollups

Rollups are sidechains [4] that batch transactions from a source chain and execute them in an external chain. Rollups are categorized as optimistic rollups (e.g., Arbitrum, Optimisim) and ZK-rollups (ZkSync, Aztec, Loopring). At the same time, optimistic rollups incur delays in transaction finality due to potential fraud challenges (up to a week). On the other hand, ZK-rollups generate proofs via a more computationally expensive method.

Typically, rollups are funded via a native bridge (e.g., Polygon proof of stake bridge, Polygon's zkEVM bridge) [138]. These bridges serve as on-ramps for layer-two technologies such as Starkware and ZkSync [139]. The latter works by parallelizing instances of a circuit implementation of the EVM, increasing the scalability of layer one blockchains. Our project would allow for on-ramping assets to sidechains with a SNARK. Our Groth16 implementation shows itself to be able to respond to the throughput necessities of bridging assets across blockchains running the EVM.

VIII. Conclusion

In this paper, we propose a strong candidate for the next-generation interoperability mechanisms: Harmonia, a framework to robustly build decentralized applications using zero-knowledge proofs. Our framework defines the existence of a set of relayers, cross-chain logic contracts, and proof verifiers. Designed with extensibility and scalability in mind, our framework does not rely on a single blockchain implementation; instead, it proves block header generation correctness on any blockchain with a light client protocol. This way, state (e.g., transaction inclusion, storage variables) on a source blockchain can be proven correct and consumed by third-party chains, allowing the realization of arbitrary cross-chain logic use cases.

As Harmonia's core, we propose DendrETH, a decentralized, secure, and efficient light client that implements Ethereum's light client sync protocol. To implement DendrETH, we leverage Circom, a circuit meta-programming language that allows us to define the light sync protocol as an arithmetic circuit. From this circuit, we create zero-knowledge proofs that attest the correct execution of the light client protocol. Our implementations are compatible with EVM-based chains and some non-EVM chains, namely chains supporting CosmWasm, and EOS. During the development of our solution, we have proposed several extensions to the light client sync protocol that 1) guarantee accountability and slashing for misbehaving parties and 2) increase the crypto-economic security of the light client.

Harmonia was thoroughly evaluated. We show that the circuit size of DendrETH translates into an acceptable throughput and latency for applications to consume: a proof can be generated in around 4 minutes. Performing updates to systems built with Harmonia costs in the order of a few thousand US dollars per year in transaction fees, a small cost to potentially host an indefinite number of cross-chain applications consuming Ethereum's state. We show that the overall latency is upper-bounded at around 25 minutes for a full end-to-end state sync. Storage-wise, storing SNARKs on-chain would accrue to around 10Mb *per annum*, yielding relatively low fees for popular Layer-2. We perform simulations on long-range attacks for light clients, yielding safety thresholds that the community can leverage. To further validate our approach, we have developed a cross-chain application that implements data transfer, using Harmonia instantiated with DendrETH.

Finally, we compiled a set of future research directions in the interoperability space, in the Appendix of this paper, that can help guide the community towards more optimal solutions, e.g., privacy-preserving bridges and cross-chain monitoring. In conclusion, Harmonia allows to design of secure, scalable, and simple cross-chain applications, hardened by the cryptography underlying SNARK technology.

Acknowledgments

We thank the open-source community for contributing to the software we used in our system. We warmly thank our colleagues in the IETF's working group Secure Asset Transfer Protocol (SATP) for fruitful discussions on interoperability security. We thank André Augusto, Guillaume Lethuillier, Freddy Zwanzger, Jan Süßenguth, Qi Feng, Peter Robinson, and Chris Spannos for providing comments that improved our paper. We thank Metacraft Labs' team, that helped implement DendrETH. We thank Dom Martinez and Iulia Mihaiu for their help in the graphical design of this paper. We thank the Nimbus team for contributing to the slashing proposal. This work was supported by the European Commission under project BIG ERA Chair (grant agreement 952226). Rafael was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and 2020.06837.BD.

References

[1] R. Belchior, J. Süßenguth, Q. Feng, T. Hardjono, A. Vasconcelos, and M. Correia, "A Brief History of Blockchain Interoperability," 6 2023. [Online]. Available: https://www.techrxiv.org/doi/full/10.36227/techrxiv.23418677.v3

[2] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, and M. Correia, "Do you need a distributed ledger technology interoperability solution?" *Distributed Ledger Technologies: Research and Practice*, vol. 2, no. 1, pp. 1–37, 2023.

[3] R. Belchior, I. Mihaiu, S. Scuri, N. Nunes, and T. Hardjono, "Towards a common standard framework for blockchain interoperability - a position paper," Tech. Rep., 2023, citation Key: belchiorrafaelCommonStandardFramework2023. [Online]. Available: https://www.techrxiv.org/users/679023/articles/678178-towards-a-common-standard-framework-for-blockchain-interoperability-a-po

[4] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–41, 2021.

[5] G. Caldarelli, “Understanding the blockchain oracle problem: A call for action,” *Information*, vol. 11, no. 11, p. 509, Oct 2020. [Online]. Available: http://dx.doi.org/10.3390/info11110509

[6] ——, “Before ethereum. the origin and evolution of blockchain oracles,” *IEEE Access*, vol. 11, pp. 50 899–50 917, 2023.

[7] M. Hargreaves, T. Hardjono, and R. Belchior, “Secure Asset Transfer Protocol (SATP),” Internet Engineering Task Force, Internet Draft draft-hargreaves-sat-core-02. [Online]. Available: https://datatracker.ietf.org/doc/draft-hargreaves-sat-core

[8] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono, “Hermes: Fault-tolerant middleware for blockchain interoperability,” *Future Generation Computer Systems*, vol. 129, pp. 236–251, 2022.

[9] T. Hardjono, A. Lipton, and A. Pentland, “Toward an Interoperability Architecture for Blockchain Autonomous Systems,” *IEEE Transactions on Engineering Management*, vol. 67, no. 4, pp. 1298–1309, 11 2020.

[10] S.-S. Lee, A. Murashkin, M. Derka, and J. Gorzny, “Sok: Not quite water under the bridge: Review of cross-chain bridge hacks,” *arXiv preprint arXiv:2210.16209*, 2022.

[11] N. Kannengießer, M. Pfister, M. Greulich, S. Lins, and A. Sunyaev, “Bridges between islands: Cross-chain technology for distributed ledger technology.”

[12] R. Belchior, P. Somogyvari, J. Pfannschmid, A. Vasconcelos, and M. Correia, “Hephaestus: Modelling, analysis, and performance evaluation of cross-chain transactions,” *TechRxiv preprint*, 2023, available at: https://tinyurl.com/muk64ve7.

[13] The Straits Times, “Cryptocurrency-bridge hacks top $1.36 billion in little over a year,” *The Straits Times*, Apr. 2022. [Online]. Available: https://www.straitstimes.com/tech/tech-news/cryptocurrency-bridge-hacks-top-136-billion-in-little-over-a-year

[14] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, and T. Hardjono, *SoK: Security and Privacy of Blockchain Interoperability*, 2023. [Online]. Available: https://www.techrxiv.org/users/687326/articles/691934-sok-security-and-privacy-of-blockchain-interoperability(more

[15] R. Browne, “$100 million worth of crypto has been stolen in another major hack,” https://www.cnbc.com/2022/06/24/hackers-steal-100-million-in-crypto-from-harmonys-horizon-bridge.html, 2022, accessed: 21-June-2023.

[16] Rekt, “Rekt - Ronin Network,” 2022. [Online]. Available: https://www.rekt.news/

[17] P. KidBold, “The Wormhole Bridge Attack Explained,” Feb. 2022. [Online]. Available: https://kaicho.substack.com/p/the-wormhole-bridge-attack-explained

[18] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, “Sok: Layer-two blockchain protocols,” in *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 2020, pp. 201–226.

[19] LiFi. (2023, Oct) LiFi’s Declassified Bridge Survival Guide. Online Blog. Available online: https://lifi.substack.com/p/lifis-declassified-bridge-survival, last accessed on 8 October 2023. [Online]. Available: https://lifi.substack.com/p/lifis-declassified-bridge-survival

[20] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, “Sok: Communication across distributed ledgers,” in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 2021, pp. 3–36.

[21] R. Belchior, “Phd thesis proposal - blockchain interoperability,” Instituto Superior Técnico, Departamento de Engenharia Informática, Tech. Rep., Sep 2021, available online: https://web.ist.utl.pt/∼ist180970/papers/phd_cat_rafael_belchior.pdf. [Online]. Available: https://web.ist.utl.pt/∼ist180970/papers/phd_cat_rafael_belchior.pdf

[22] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, “Flyclient: Super-light clients for cryptocurrencies,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 928–946.

[23] E. Foundation, “The merge,” https://ethereum.org/en/roadmap/merge/, 2023, accessed: 21-June-2023.

[24] E. Burger, B. Chiang, S. Chokshi, E. Lazzarin, J. Thaler, and A. Yahya, “The zero knowledge canon,” https://a16zcrypto.com/posts/article/zero-knowledge-canon/, 2023, accessed: 6-July-2023.

[25] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof-systems,” in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 203–225.

[26] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, “From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again,” in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012, pp. 326–349.

[27] R. Belchior, L. Torres, J. Pfannschmid, A. Vasconcelos, and M. Correia, “Can we share the same perspective? blockchain interoperability with views,” Oct 2022. [Online]. Available: https://www.techrxiv.org/articles/preprint/Is_My_Perspective_Better_Than_Yours_Blockchain_Interoperability_with_Views/20025857/3

[28] H. Liu, X. Luo, H. Liu, and X. Xia, “Merkle tree: A fundamental component of blockchains,” in *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*. IEEE, 2021, pp. 556–561.

[29] M. Borkowski, C. Ritzer, D. McDonald, and S. Schulte. (2018) Caught in chains: Claim-first transactions for cross-blockchain asset transfers. Technische Universität Wien. Available online: https://www.researchgate.net/profile/Michael-Borkowski/publication/327364072_Caught_in_Chains_Claim-First_Transactions_for_Cross-Blockchain_Asset_Transfers/links/5c656f3a92851c48a9d3b929/Caught-in-Chains-Claim-First-Transactions-for-Cross-Blockchain-Asset-Transfers.pdf (Accessed on 16 October 2023). [Online]. Available: https://www.researchgate.net/profile/Michael-Borkowski/publication/327364072_Caught_in_Chains_Claim-First_Transactions_for_Cross-Blockchain_Asset_Transfers/links/5c656f3a92851c48a9d3b929/Caught-in-Chains-Claim-First-Transactions-for-Cross-Blockchain-Asset-Transfers.pdf

[30] A. A. Domenech, J. Heiss, and S. Tai, “Servicifying zk-snarks execution for verifiable off-chain computations,” 2024.

[31] M. Graf, R. Küsters, and D. Rausch, “Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric,” in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 236–255.

[32] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, “zkbridge: Trustless cross-chain bridges made practical,” *arXiv preprint arXiv:2210.00264*, 2022.

[33] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[34] M. Iqbal and R. Matulevičius, “Exploring sybil and double-spending risks in blockchain systems,” *IEEE Access*, vol. 9, pp. 76 153–76 177, 2021.

[35] R. Belchior, L. Torres, J. Pfannschmid, A. Vasconcelos, and M. Correia, “Can We Share the Same Perspective? Blockchain Interoperability with Views,” 10 2022. [Online]. Available: https://www.techrxiv.org/articles/preprint/Is_My_Perspective_Better_Than_Yours_Blockchain_Interoperability_with_Views/20025857

[36] S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk *et al.*, “Cryptographic hash functions: A survey,” Citeseer, Tech. Rep., 1995.

[37] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, “Aggregate and verifiably encrypted signatures from bilinear maps,” in *Advances in Cryptology—EUROCRYPT 2003: International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4–8, 2003 Proceedings 22*. Springer, 2003, pp. 416–432.

[38] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the weil pairing,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2001, pp. 514–532.

[39] Eth2Book, “Altair: Part 2 - building blocks - signatures,” https://eth2book.info/altair/part2/building_blocks/signatures/, 2023, accessed: 27-June-2023.

[40] I. Ozcelik, S. Medury, J. Broaddus, and A. Skjellum, “An overview of cryptographic accumulators,” *arXiv preprint arXiv:2103.04330*, 2021.

[41] F. Baldimtsi, J. Camenisch, M. Dubovitskaya, A. Lysyanskaya, L. Reyzin, K. Samelin, and S. Yakoubov, “Accumulators with applications to anonymity-preserving revocation,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 301–315.

[42] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” https://bitcoin.org/bitcoin.pdf, 2008, accessed: 21-June-2023.

[43] P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, “Sok: Blockchain light clients,” in *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 2022, pp. 615–641.

[44] M. Bellare and O. Goldreich, “On defining proofs of knowledge,” in *Annual International Cryptology Conference*. Springer, 1992, pp. 390–420.

[45] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, “Bulletproofs: Short Proofs for Confidential Transactions and More,” *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2018-May, pp. 315–334, 2018, iSBN: 9781538643525.

[46] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, “A survey on zero-knowledge proof in blockchain,” *IEEE network*, vol. 35, no. 4, pp. 198–205, 2021.

[47] J. Groth, “On the size of pairing-based non-interactive arguments,” in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 305–326.

[48] ——, “Simulation-sound nizk proofs for a practical language and constant size group signatures,” in *Advances in Cryptology–ASIACRYPT 2006: 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006. Proceedings 12*. Springer, 2006, pp. 444–459.

[49] S. Hu, M. Li, J. Weng, J.-N. Liu, J. Weng, and Z. Li, “Ivyredaction: Enabling atomic, consistent and accountable cross-chain rewriting,” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2023.

[50] A. Odlyzko, “Discrete logarithms: The past and the future,” *Towards a Quarter-Century of Public Key Cryptography: A Special Issue of DESIGNS, CODES AND CRYPTOGRAPHY An International Journal. Volume 19, No. 2/3 (2000)*, pp. 59–75, 2000.

[51] G. Leurent and P. Q. Nguyen, “How risky is the random-oracle model?” in *Annual International Cryptology Conference*. Springer, 2009, pp. 445–464.

[52] M. Petkus. (2019) Why and how zk-snark works. Available online: https://arxiv.org/pdf/1906.07221.pdf, Accessed on 16 October 2023.

[53] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer, “The hunting of the snark,” *Journal of Cryptology*, vol. 30, no. 4, pp. 989–1066, 2017.

[54] A. Nitulescu. (2020) zk-snarks: a gentle introduction. Available online: https://www.di.ens.fr/~nitulesc/files/Survey-SNARKs.pdf, Accessed on 16 October 2023.

[55] K. Qin, L. Zhou, and A. Gervais, “Quantifying blockchain extractable value: How dark is the forest?” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 198–214.

[56] C. McMenamin, “Sok: Cross-domain mev,” *arXiv preprint arXiv:2308.04159*, 2023.

[57] A. Obadia, A. Salles, L. Sankar, T. Chitra, V. Chellani, and P. Daian, “Unity is strength: A formalization of cross-domain maximal extractable value,” *arXiv preprint arXiv:2112.01472*, 2021.

[58] R. Belchior. (2023, Sep) Dlt interoperability and more 28 — sok: Cross-domain mev. Accessed on 13 September 2023. [Online]. Available: https://rafaelbelchior.medium.com/dlt-interoperability-and-more-%EF%B8%8F-28-sok-cross-domain-mev-%EF%B8%8F-477971a4887e

[59] Ethereum, “Altair sync protocol - annotated spec,” https://github.com/ethereum/annotated-spec/blob/master/altair/sync-protocol.md, 2023, accessed: 30-June-2023.

[60] D. H. Staff, “What does altair bring to ethereum 2.0?” https://dailyhodl.com/2021/11/04/what-does-altair-bring-to-ethereum-2-0/, 2021, accessed: 27-June-2023.

[61] t3rn, “Exploring Eth’s Altair Light Client Protocol,” 2023, available online: https://www.t3rn.io/blog/exploring-eths-altair-light-client-protocol-t3rns-vision, last accessed on 2023-09-18. [Online]. Available: https://www.t3rn.io/blog/exploring-eths-altair-light-client-protocol-t3rns-vision

[62] J. Prestwich, “Altair,” https://prestwich.substack.com/p/altair, 2023, accessed: 29-July-2023.

[63] Flashbots, “Flashbots documentation,” https://docs.flashbots.net/, 2023, accessed: 29-July-2023.

[64] Nimbus team. (2023) Sync committee slashing · Issue #3321 · ethereum/consensus-specs. Available online: https://github.com/ethereum/consensus-specs/issues/3321, last accessed on [your-access-date]. [Online]. Available: https://github.com/ethereum/consensus-specs/issues/3321

[65] Ethereum Foundation. (2023) Beacon Chain Specification. Available online: https://github.com/ethereum/consensus-specs/blob/dev/specs/altair/beacon-chain.md, last accessed on 2023-09-18. [Online]. Available: https://github.com/ethereum/consensus-specs/blob/dev/specs/altair/beacon-chain.md

[66] E. N. Tas and D. Boneh, “Cryptoeconomic security for data availability committees,” no. arXiv:2208.02999, Jun 2023, arXiv:2208.02999 [cs]. [Online]. Available: http://arxiv.org/abs/2208.02999

[67] M. Westerkamp and A. Küpper, “Smartsync: Cross-blockchain smart contract interaction and synchronization,” in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, May 2022, p. 1–9.

[68] E. Fynn, A. Bessani, and F. Pedone, “Smart contracts on the move,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 233–244.

[69] H. D. Bandara, X. Xu, and I. Weber, “Patterns for blockchain data migration,” in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–19.

[70] Infura, “Blockchain infrastructure for the new internet,” https://www.infura.io/, 2023, accessed: 10-July-2023.

[71] Alchemy, “The web3 development platform,” https://www.alchemy.com/, 2023, accessed: 10-July-2023.

[72] Blockdaemon, “Powerful access to indexed multi-chain data in seconds,” https://www.blockdaemon.com/nodes/universal-api, 2023, accessed: 10-July-2023.

[73] Figment, “Figment: Simplified blockchain infrastructure and tools,” https://figment.io/, 2023, accessed: 10-July-2023.

[74] iden3, “snarkjs: a pure javascript zksnark library,” https://github.com/iden3/snarkjs, 2023, accessed: 27-June-2023.

[75] Contributors. (2023) Foundry: A toolkit for ethereum application development. Accessed: 13 September 2023. [Online]. Available: https://github.com/foundry-rs

[76] (2023, Sep) Circom documentation. Accessed on 15 September 2023. [Online]. Available: https://docs.circom.io/

[77] “The nimbus guide,” Sep 2023, accessed on 15 September 2023. [Online]. Available: https://nimbus.guide/

[78] “Documentation for the go-ethereum client,” Sep 2023, accessed on 15 September 2023. [Online]. Available: https://geth.ethereum.org/docs

[79] (2023, Mar) Bor v0.3.7 - mainnet and mumbai release. Polygon Community Forum. Accessed on 15 September 2023. [Online]. Available: https://forum.polygon.technology/t/bor-v0-3-7-mainnet-and-mumbai-release/11559

[80] FluiDex. (2023, Apr) snarkit2: A toolkit to compile and debug circom circuit. GitHub repository. Accessed on 13 September 2023. [Online]. Available: https://github.com/fluidex/snarkit2

[81] E. Foundation, “Explanation of single slot finality,” https://ethereum.org, 2023, accessed: 21-June-2023.

[82] M. Straka, “Recursive zero-knowledge proofs: A comprehensive primer,” https://www.michaelstraka.com/recursive-zero-knowledge-proofs, 2023, accessed: 2024-05-28.

[83] 0xPARC, “Groth16 recursion,” https://0xparc.org/blog/groth16-recursion, 2023, accessed: 2024-05-28.

[84] A. Chiesa, R. Lehmkuhl, P. Mishra, and Y. Zhang, “Eos: Efficient private delegation of zksnark provers,” in *USENIX Security Symposium. USENIX Association*, 2023.

[85] J. R. F. Cacho and K. Taghva, “The state of reproducible research in computer science,” in *17th International Conference on Information Technology–New Generations (ITNG 2020)*. Springer, 2020, pp. 519–524.

[86] J. Cito and H. C. Gall, “Using docker containers to improve reproducibility in software engineering research,” in *Proceedings of the 38th international conference on software engineering companion*, 2016, pp. 906–907.

[87] M. Labs, “Issue 3321 from ethereum proof-of-stake consensus specifications,” https://github.com/ethereum/consensus-specs/issues/3321, 2023, accessed: 29-July-2023.

[88] V. Buterin, “Proof of stake: How i learned to love weak subjectivity,” https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/, 2014, accessed: 27-June-2023.

[89] A. Asgaonkar. (2020) Weak subjectivity in eth2.0. Available online: https://notes.ethereum.org/@adiasg/weak-subjectvity-eth2, last accessed on [your-access-date]. [Online]. Available: https://notes.ethereum.org/@adiasg/weak-subjectvity-eth2

[90] M. Labs. (2023) Safety considerations for long-range ethereum light client syncing. Available

online: https://github.com/metacraft-labs/DendrETH/tree/main/docs/long-range-syncing#sync_committee_slashing-proposal, last accessed on 2023-09-18. [Online]. Available: https://github.com/metacraft-labs/DendrETH/tree/main/docs/long-range-syncing#sync_committee_slashing-proposal

[91] R. Belchior, M. Correia, A. Augusto, and T. Hardjono, "Satp gateway crash recovery mechanism," IETF, Technical Report, 2023, accessed: 29-July-2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-belchior-satp-gateway-recovery/

[92] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," *arXiv preprint arXiv:1904.05234*, 2019.

[93] Etherscan. Ethereum blocks on etherscan. Accessed on 5th October 2023. [Online]. Available: https://etherscan.io/blocks

[94] M. Vasek, M. Thornton, and T. Moore, "Empirical analysis of denial-of-service attacks in the bitcoin ecosystem," in *Financial Cryptography and Data Security*, R. Böhme, M. Brenner, T. Moore, and M. Smith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 57–71.

[95] T. Verge, "Nomad bridge chaotic hack leads to $200 million in cryptocurrency losses," https://www.theverge.com/2022/8/2/23288785/nomad-bridge-200-million-chaotic-hack-smart-contract-cryptocurrency, 2022, accessed: 29-July-2023.

[96] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *IACR Cryptology ePrint Archive*, vol. 2018, p. 046, 2018.

[97] T. Z. Team, "Multi-party computation for zcash," 2017, accessed: 2023-06-21. [Online]. Available: https://github.com/zcash/mpc/blob/master/whitepaper.pdf

[98] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," Cryptology ePrint Archive, Paper 2017/1050, 2017, https://eprint.iacr.org/2017/1050. [Online]. Available: https://eprint.iacr.org/2017/1050

[99] Y. Yu and X. Xie, "Privacy-preserving computation in the post-quantum era," *National Science Review*, vol. 8, no. 9, p. nwab115, 2021.

[100] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[101] R. Steinfeld, "Post-quantum zero-knowledge proofs and applications," in *Proceedings of the 10th ACM Asia Public-Key Cryptography Workshop*, ser. APKC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1. [Online]. Available: https://doi.org/10.1145/3591866.3593075

[102] Y. Ishai, H. Su, and D. J. Wu, "Shorter and faster post-quantum designated-verifier zksnarks from lattices," Cryptology ePrint Archive, Paper 2021/977, 2021, https://eprint.iacr.org/2021/977. [Online]. Available: https://eprint.iacr.org/2021/977

[103] A. Berentsen, J. Lenzi, and R. Nyffenegger, "A walk-through of a simple zk-stark proof," no. 4308637, Dec 2022. [Online]. Available: https://papers.ssrn.com/abstract=4308637

[104] A. Vlasov, K. Olson, and A. Stokes. (2020) Eip-2537: Precompile for bls12-381 curve operations. Available online: https://eips.ethereum.org/EIPS/eip-2537, last accessed on 2023-09-21. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2537

[105] T. Groves and J. Ledyard, "Optimal allocation of public goods: A solution to the" free rider" problem," *Econometrica: Journal of the Econometric Society*, pp. 783–809, 1977.

[106] ConsenSys, "Frontrunning," https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning/, 2023, accessed: 10-July-2023.

[107] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "Xclaim: Trustless, interoperable, cryptocurrency-backed assets," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 193–210.

[108] V. Buterin and V. Griffith, "Casper the friendly finality gadget," no. arXiv:1710.09437, Jan. 2019, arXiv:1710.09437 [cs]. [Online]. Available: http://arxiv.org/abs/1710.09437

[109] M. Labs, "Dendreth casper finality proofs," Sep 2023, available online: https://hackmd.io/@metacraft-labs/DendrETH-Casper-Finality-Proofs, last accessed on 29 September 2023. [Online]. Available: https://hackmd.io/@metacraft-labs/DendrETH-Casper-Finality-Proofs

[110] Bhuptani, Arjun, "The Interoperability Trilemma: AKA Why Bridging Ethereum Domains is So Damn Difficult," 2021, available online: https://blog.connext.network/the-interoperability-trilemma-657c2cf69f17, last accessed on 2023-05-21. [Online]. Available: https://blog.connext.network/the-interoperability-trilemma-657c2cf69f17

[111] G. Caldarelli and J. Ellul, "The blockchain oracle problem in decentralized finance—A multivocal approach," *Applied Sciences (Switzerland)*, vol. 11, no. 16, 2021.

[112] Zarick, Ryan and Pellegrino, Bryan and Banister, Caleb, "LayerZero: Trustless Omnichain Interoperability Protocol," 2021, available online: https://layerzero.network/pdf/LayerZero_Whitepaper_Release.pdf, last accessed on 2023-05-22. [Online]. Available: https://layerzero.network/pdf/LayerZero_Whitepaper_Release.pdf

[113] B. ONeill, D. Hyland-Wood, E. Abebe, J. Bedi, K. Adams, M. Quintyne-Collins, P. Robinson, R. Chen, and S. Casey, "Bridge assessment report," https://uniswap.notion.site/Bridge-Assessment-Report-0c8477afadce425abac9c0bd175ca382, 2023, accessed: 21-June-2023.

[114] L2Beat, "L2beat: Bridges overview," https://l2beat.com/bridges/summary, 2023, accessed: 29-July-2023.

[115] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1299–1316.

[116] R. Belchior, A. Castaño, and P. Somogyvari, "Htlcs in hyperledger cacti," https://github.com/hyperledger/cacti/tree/main/packages/cactus-plugin-htlc-eth-besu, 2023, accessed: 29-July-2023.

[117] D. Gruber, W. Li, and G. Karame, "Unifying lightweight blockchain client implementations," in *Proc. NDSS workshop decentralized IoT security stand*, 2018, pp. 1–7.

[118] M. Al-Bassam, A. Sonnino, and V. Buterin, "Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities," *arXiv preprint arXiv:1809.09044*, vol. 160, 2018.

[119] S. Paavolainen and C. Carr, "Security properties of light clients on the ethereum blockchain," *IEEE Access*, vol. 8, pp. 124 339–124 358, 2020.

[120] S. Braithwaite, E. Buchman, I. Khoffi, I. Konnov, Z. Milosevic, R. Ruetschi, and J. Widder, "A tendermint light client," no. arXiv:2010.07031, Oct 2020, arXiv:2010.07031 [cs]. [Online]. Available: http://arxiv.org/abs/2010.07031

[121] C. Goes, "The interblockchain communication protocol: An overview," no. arXiv:2006.15918, Jun 2020, arXiv:2006.15918 [cs]. [Online]. Available: http://arxiv.org/abs/2006.15918

[122] I. Team, "Btc-relay specification," https://spec.interlay.io/spec/btc-relay/index.html, 2023, accessed: 27-June-2023.

[123] X. Wang, S. Duan, J. Clavin, and H. Zhang, "Bft in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–37, 2022.

[124] E. Androulaki, A. Barger, V. Bortnikov, S. Muralidharan, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Murthy, C. Ferris, G. Laventman, Y. Manevich, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, vol. 2018-Janua. New York, New York, USA: Association for Computing Machinery, Inc, Apr 2018, p. 1–15, citation Key: fabric.

[125] C. Pedreira, R. Belchior, M. Matos, and A. Vasconcelos, "Securing asset transfers on permissioned blockchains," in *Proceedings of the BlockTEE 2022 Workshop*, 2022, accessed: 27-June-2023. [Online]. Available: https://www.techrxiv.org/articles/preprint/Trustable_Blockchain_Interoperability_Securing_Asset_Transfers_on_Permissioned_Blockchains/19651248

[126] E. Abebe, Y. Hu, A. Irvin, D. Karunamoorthy, V. Pandit, V. Ramakrishna, and J. Yu, "Verifiable observation of permissioned ledgers," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–9.

[127] Chand, Arjun, "What Are Blockchain Bridges And How Can We Classify Them? Classifying Bridges As We Know Them," 2022. [Online]. Available: https://tinyurl.com/bdfn4x4s

[128] Bloomberg, "Hackers steal about 600 million in one of the biggest crypto heists," *Bloomberg.com*, Mar 2022. [Online]. Available: https://www.bloomberg.com/news/articles/2022-03-29/hackers-steal-590-million-from-ronin-in-latest-bridge-attack

[129] Y. Xue and M. Herlihy, "Hedging against sore loser attacks in cross-chain transactions," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 155–164.

[130] C. Faife, "Explaining crypto's billion-dollar bridge problem," Apr 2022. [Online]. Available: https://www.theverge.com/23017107/crypto-billion-dollar-bridge-hack-decentralized-finance

[131] M. Westerkamp and J. Eberhardt, "zkrelay: Facilitating sidechains using zksnark-based chain-relays," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSPW)*, Sep. 2020, p. 378–386, citation Key: westerkamp_zkrelay_2020.

[132] S. Labs, "Telepathy documentation," https://docs.telepathy.xyz/, 2023, accessed: 29-July-2023.

[133] (2023) Herodotus documentation. Available online: https://docs.herodotus.dev/herodotus-docs/, last accessed on 2023-09-18. [Online]. Available: https://docs.herodotus.dev/herodotus-docs/

[134] M. Westerkamp and J. Eberhardt, "zkrelay: Facilitating sidechains using zksnark-based chain-relays," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020, pp. 378–386.

[135] bunny sleepy, "private-bridge-zcash-ethereum," Jun 2023. [Online]. Available: https://github.com/bunny-sleepy/private-bridge-zcash-ethereum

[136] Webb, "Overview - anchor system," 2023, accessed: 29-July-2023. [Online]. Available: https://docs.webb.tools/docs/concepts/anchor-system/overview/

[137] O. Ciobotaru, F. Shirazi, A. Stewart, and S. Vasilyev, "Accountable light client systems for pos blockchains," no. 2022/1205, 2022, publication info: Preprint. [Online]. Available: https://eprint.iacr.org/2022/1205

[138] P. p. a. team, "Polygon bridge: Bridge assets from ethereum to polygon zkevm." [Online]. Available: https://wallet.polygon.technology/

[139] u. family=labs.io, prefix=https://matter. Welcome to our Docs — zkSync Era. Welcome to our Docs — zkSync Era Docs. [Online]. Available: https://era.zksync.io/docs/

[140] V. Buterin, "Ethereum whitepaper: A next-generation smart contract and decentralized application platform," https://ethereum.org/en/whitepaper/, 2013, accessed: 21-June-2023.

[141] F. M. Benčić and I. P. Žarko, "Distributed ledger technology: Blockchain compared to directed acyclic graph," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1569–1570.

[142] W. Al-Saqaf and N. Seidler, "Blockchain technology for social impact: opportunities and challenges ahead," *Journal of Cyber Policy*, vol. 2, no. 3, pp. 338–354, 2017.

[143] R. Belchior, A. Vasconcelos, and M. Correia, "Towards secure, decentralized, and automatic audits with blockchain," in *European Conference on Information Systems*, 2020, citation Key: Belchior2020ecis.

[144] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono, "Enabling cross-jurisdiction digital asset transfer," in *IEEE International Conference on Services Computing*. IEEE, 2021, citation Key: belchior-hermes-ieeescc.

[145] D. A. Zetzsche, D. W. Arner, and R. P. Buckley, "Decentralized finance (defi)," *Journal of Financial Regulation*, vol. 6, pp. 172–203, 2020.

[146] K. Qin, L. Zhou, Y. Afonin, L. Lazzaretti, and A. Gervais, "Cefi vs. defi–comparing centralized to decentralized finance," *arXiv preprint arXiv:2106.08157*, 2021.

[147] R. Belchior, B. Putz, G. Pernul, M. Correia, A. Vasconcelos, and S. Guerreiro, "Ssibac: self-sovereign identity based access control," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 1935–1943.

[148] J. Ernstberger, J. Lauinger, F. Elsheimy, L. Zhou, S. Steinhorst, R. Canetti, A. Miller, A. Gervais, and D. Song, "Sok: Data sovereignty," *Cryptology ePrint Archive*, 2023.

[149] P. Dutta, T.-M. Choi, S. Somani, and R. Butala, "Blockchain technology in supply chain operations: Applications, challenges and research opportunities," *Transportation research part e: Logistics and transportation review*, vol. 142, p. 102067, 2020.

[150] B. Putz, M. Dietz, P. Empl, and G. Pernul, "Ethertwin: Blockchain-based secure digital twin information management," *Information Processing & Management*, vol. 58, no. 1, p. 102425, 2021, citation Key: PUTZ2021102425.

[151] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE, 2018, pp. 1–4.

[152] M. Correia, "From Byzantine Consensus to Blockchain Consensus," *Essentials of Blockchain Technology*, p. 41, 2019.

[153] M. Raikwar, D. Gligoroski, and K. Kralevska, "Sok of used cryptography in blockchain," *IEEE Access*, vol. 7, pp. 148 550–148 575, 2019.

[154] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, T. Takeuchi, T. Kuhrt, and R. Belchior, "Hyperledger cactus whitepaper," Hyperledger, Tech. Rep. 2. [Online]. Available: https://github.com/hyperledger/cactus/blob/main/docs/whitepaper/whitepaper.md

[155] E. Foundation, "Introduction to merkle patricia trie," https://ethereum.org, 2023, accessed: 21-June-2023.

[156] J. Chiang. (2023) The beacon chain ethereum 2.0 explainer you need to read first. [Online]. Available: https://ethos.dev/beacon-chain

[157] bitfly gmbh, "Beacon chain: Ethereum 2.0 block explorer," https://www.beaconcha.in/, 2023, accessed: 21-June-2023.

[158] "Simplified Active Validator Cap and Rotation Proposal - Proof-of-Stake," Mar 2021, available online: https://ethresear.ch/t/simplified-active-validator-cap-and-rotation-proposal/9022, last accessed on 8 October 2023. [Online]. Available: https://ethresear.ch/t/simplified-active-validator-cap-and-rotation-proposal/9022

[159] P. Gaži, A. Kiayias, and D. Zindros, "Proof-of-stake sidechains," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 139–156.

[160] S. Agrawal, J. Neu, E. N. Tas, and D. Zindros, "Proofs of proof-of-stake with sublinear complexity," *arXiv preprint arXiv:2209.08673*, 2022.

[161] A. Berentsen, J. Lenzi, and R. Nyffenegger, "An introduction to zero-knowledge proofs in blockchains and economics," *Federal Reserve Bank of St. Louis Review*, 2023.

[162] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs," in *Theory of Cryptography: 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*. Springer, 2013, pp. 315–333.

[163] A. Chiesa, R. Lehmkuhl, P. Mishra, and Y. Zhang, "Eos: Efficient private delegation of zksnark provers."

[164] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[165] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 2010, pp. 177–194.

[166] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Advances in Cryptology—CRYPTO'86: Proceedings 6*. Springer, 1987, pp. 186–194.

[167] zkSecurity, "The nova attack," https://www.zksecurity.xyz/blog/posts/nova-attack/, 2023, accessed: 29-July-2023.

[168] B. O'Neill, D. Hyland-Wood, E. Abebe, J. Bedi, K. Adams, M. Quintyne-Collins, P. Robinson, R. Chen, and S. Casey. (2023, Sep) Bridge assessment report. Accessed on 15 September 2023. [Online]. Available: https://uniswap.notion.site/Bridge-Assessment-Report-0c8477afadce425abac9c0bd175ca382

[169] ConsenSys, "Consensys announces the beta release of metamask bridges in the portfolio dapp," https://tinyurl.com/3y5the44, 2023, accessed: 29-July-2023.

[170] OpenIBC. (2023) Openibc. Available online: https://www.openibc.com/, last accessed on 2023-09-18. [Online]. Available: https://www.openibc.com/

[171] Cosmos. (2023) Inter-blockchain communication protocol (ibc) by cosmos. Available online: https://github.com/cosmos/ibc, last accessed on 2023-09-18. [Online]. Available: https://github.com/cosmos/ibc

[172] B. Asselstine, P. Turelier, and C. Whinfrey. (2022) Eip-5164. Available online: https://eips.ethereum.org/EIPS/eip-5164, last accessed on 2023-09-18. [Online]. Available: https://eips.ethereum.org/EIPS/eip-5164

[173] G. Karame and S. Capkun, "Blockchain security and privacy," *IEEE Security & Privacy*, vol. 16, no. 04, pp. 11–12, 2018.

[174] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[175] R. Zhang, R. Xue, and L. Liu, "Security and privacy on blockchain," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 1–34, 2019.

[176] Y. Gai, L. Zhou, K. Qin, D. Song, and A. Gervais, “Blockchain large language models,” *arXiv preprint arXiv:2304.12749*, 2023.

APPENDIX A
ADDITIONAL CONTEXT

The past years have seen significant advances in the art of decentralized storage and computation, foundations for blockchains, and distributed ledger technology (DTL) [42], [124], [140], [141]. DLTs are networks of peers that maintain a replicated database and achieve consensus on that database (so-called ledger) without any trusted intermediary. In 2023, blockchain technology remains paramount, its value clearly demonstrated by the market capitalization of cryptocurrencies, which stands around US$835 billion, even after a significant decline from its peak at the end of 2021. This significant market value underscores the global acceptance and reliance on blockchain technology, underpinning the operation of cryptocurrencies and other relevant use cases, such as social impact [142], [143], decentralized finance [144]–[146], decentralized identity [147], [148], process optimization [149], [150], and many more [151]. Indeed, the development of blockchain technologies gave back to traditional distributed systems [152] and cryptography research [153], highlighting the importance of this new discipline.

Counting with hundreds of DLTs and thousands of cryptocurrencies, with their own design decisions (privacy vs. solubility vs. decentralization vs. security), security models (crash fault-tolerant vs. Byzantine fault-tolerant), consensus algorithms (proof of work, proof of stake, proof of X), and specific implementations, different technologies are inherently and inevitably heterogeneous, leading to a service market that is diverse in features and trade-offs. The field of blockchain interoperability became a hot topic when researchers and practitioners alike began to explore the qualities of different infrastructures and use them together [2]. Thus, there are attempts to answer the question: how can a process in a source domain realize a business workflow that is separated across different centralized and decentralized systems, such that there is a unified view of the state over these different systems [35]. Cross-chain logic defines the rules that guide a set of local transactions in different systems, given variables such as transaction inclusion in a certain system, timestamps of such transactions, and so on. For example, the cross-chain logic of a simple bridge definition would be only mint tokens on the target blockchain for the user if the user locked an equivalent amount of tokens on the source blockchain [12].

The case for the need for blockchain interoperability is not new. Indeed, one can interpret the problem of blockchain interoperability as achieving ACID properties (atomicity, consistency, isolation, and durability) for transactions taking place on an abstract distributed database with a centralized or decentralized controller [8]. Several cross-chain use cases emerge, enabled by tools and monitoring frameworks [12], and supporting processes, such as blockchain migration [154], state migration [67], and cross-jurisdiction asset transfers [144].

APPENDIX B
MERKLE PROOF VERIFICATION

The validity of the proof can be computed (algorithm `verify`) in the following way (hash is a secure cryptographic hash that is preimage and collision-resistant, and j is a flag to decide on the order of concatenation, depending if the next node is a left or right child):

Algorithm 6: Merkle Tree Proof Verification (algorithm `verify`)

```
Input: $\pi_i$, $root$, $v[i]$
Output: $\{0, 1\}$
currentHash = Hash($v[i]$)
foreach $\pi_{i,j}$ in $\pi_i$ do
    if $j$ is even then
        currentHash = Hash($\pi_{i,j} + currentHash$)
    end if
    else
        currentHash = Hash($currentHash + \pi_{i,j}$)
    end if
end foreach
if currentHash == root then
    return 1
else
    return 0
end if
```

To verify the proof, the verifier recomputes the hashes along the path from the root of the tree to the leaf node (or the other way around), using the provided nodes in the proof. If the computed hash of the leaf node matches the expected hash, the proof is valid, and the key-value pair represented by a leaf node exists in the Merkle tree. Otherwise, the proof is invalid (badly constructed proof or the data is not included in the tree). The verifier time and proof size and logarithmic in the number of elements of the tree. One can prove non-inclusion of an element by supplying merkle proofs for the closest keys that enclose the missing one (cf. Figure 12).

APPENDIX C
THE ETHEREUM BLOCKCHAIN

Ethereum, launched in 2015, is the first blockchain designed to enable smart contracts and decentralized applications (DApps). As of June 2023, Ethereum is the second biggest blockchain in terms of market capitalization, and the biggest blockchain with support for Turing complete smart contracts.

1) System Actors: Ethereum has three types of nodes: full nodes, archival nodes, and light nodes. Full nodes verify the state of the blockchain, namely its accounts, but prune accounts state trees that are older than 1024 blocks. Archival nodes do not prune any account trees and keep the whole history of the blockchain. Light nodes only store block headers and are ideal to be run in resource-constrained environments. Validators are nodes that contribute to the consensus of the network and earn rewards, by staking a certain amount of Eth

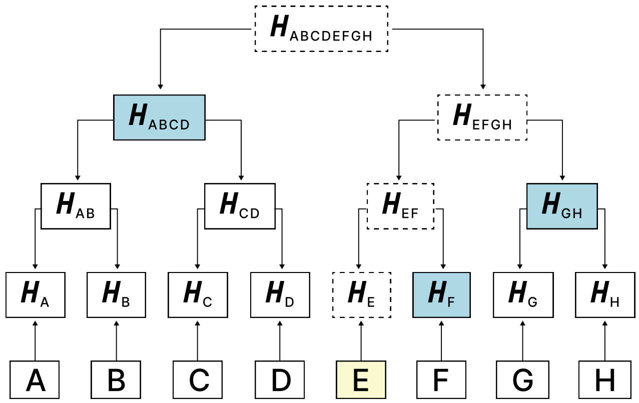


Fig. 12: Merkle proof example. Proof of inclusion for data item E needs H_F, H_{GH}, and H_{ABCD}. The verification algorithm would hash E and concatenate it with H_F, concatenate H_{EF} with H_{GH} and hash the result; concatenate H_{EFGH} with H_{ABCD}, hash it, and compare it with the root.

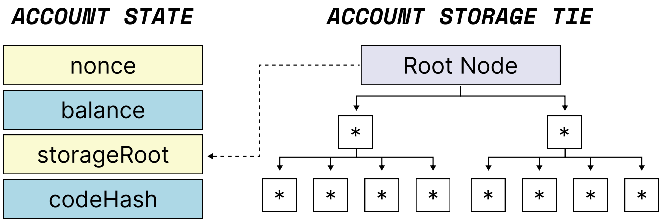


Fig. 13: Account state encoding in Ethereum

that can be slashed in case of misbehaviour. For Ethereum nodes to become validators on the network (full nodes or archival nodes), they need to stake 32 Eth (approximately 59,200 USD at the time of writing). Ethereum node software implementations include client and wallet functionalities (used interchangeably), allowing users to sign transactions that are then picked by a node and broadcasted to the network. We are specifically interested in the light clients implemented as smart contracts for interoperability.

2) State: Ethereum is an account-based blockchain. Externally-owned accounts are controlled by anyone with the private keys referring to that account. On the other hand, contract accounts are smart contracts that contain business logic that is callable by transactions (either from externally owned accounts or other smart contracts). The state of all accounts is represented with a Merkle tree, where its root (also called state root, which is stored in each block header) is updated for every transaction (one can see the state root being changed at every block using a block explorer [22]). This Merkle tree maps between addresses to account states and is called the world state tree (cf. Figure 13).

In more detail, Ethereum uses a version of the Merkle tree called Merkle Patricia Tree (MPT) [155] to store the world state. MPTs are hexary tree, and have some some performance optimizations versus Merkle trees (i.e., extension nodes for path compression). The mechanism to generate Merkle proofs in a Patricia tree is equivalent to the mechanism to generate binary trees. Instead of hashing pairs of nodes, the algorithm hashes the nodes included in the path, which could be a variable number.

Each account in Ethereum has a few attributes, e.g., nonce, balance, storage root, and storage. The storage root is the hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (which is a mapping between 32-byte keys and 32-byte values; where each pair is called a storage slot). This is only used in contract accounts and not in EOAs. Each storage modification in a contract updates this hash. The storage, on its hand, has a Merkle Patricia Tree that stores the contract state (the mappings referred above).

The root hash of this storage tree is what gets included in the account state as the storage root. A change to the storage will change the storage tree root, which in turn will change the account's Merkle root. Consequently, this causes a change to the state root. Essentially, this process assures all state transitions are valid and that the entire blockchain state is tamper-evident.

Knowing the data structure of Ethereum is stored as a set of Merkle trees, meaning that we can prove anything from the global state using one or more Merkle proofs. In practice, to prove an account exists (or it has a certain nonce, or balance), one can construct a Merkle (Patricia) proof that consists of the path that includes all the tree nodes from the state root to the leaf node that stores the account state. Similarly, to prove the storage of an account, one can provide a merkle proof that accounts for a storage key, i.e., the Merkle proof path between the storage key and the storage hash; and the Merkle proof that the account which the storage hash refers to exists (above proof). Finally, to prove a transaction is included in a block, one can create a Merkle proof against the transactions tree that is a path from the leaf containing the transaction to the transaction root; and a Merkle proof that the transactions root is included in the block (using the block root).

A. Consensus

Ethereum has moved away from the classical proof of work consensus, where each validator executed an expensive algorithm, *Ethash*. Solving an iteration of Ethash would grant the right to propose the next block. In the current Ethereum specification, the consensus is proof of stake. In proof of stake, block proposers are validators randomly chosen to build a block, proportional to the amount of the currency they hold in the blockchain (namely Eth; therefore, block proposers have a stake in the network).

Currently, there are around 21 million ETH staked[23] (around

---

[22] for example, see Etherscan.

[23] as per the source stakingrewards.com.

40B USD), making Ethereum a very secure blockchain from a cryptoeconomics perspective. To stake ETH and earn rewards, validators lock 32 ETH, which will be unavailable. For validators to withdraw their assets, they must enter an exit queue (after serving at least 2,048 epochs, around nine days [156]). The queue implements a churn limit [65] (the minimum rate is four validators entering plus leaving per epoch) that specifies the maximum number of validators that can join or exit the validator set at any epoch- The churn rate aims to prevent a large portion of malicious validators from performing some malicious action and then immediately leaving to escape being slashed.

Ethereum's global clock uses the notion of slots and epochs. Each slot is 12 seconds, and an epoch of 32 slots is 6.4 minutes. At every epoch, validators are evenly divided across slots and then subdivided into committees of appropriate size. Each slot has committees of at least 128 validators. All of the validators from that slot attest to the Beacon Chain head. An attacker has a negligible probability of controlling a supermajority of a committee (considering that 128 validators are randomly picked from hundreds of thousands of nodes). This security comes from the fact that an attacker cannot predict which validators will be chosen for each slot (and thus be able to bribe them beforehand). To solve this problem, at the beginning of every epoch, a pseudorandom process called RANDAO selects proposers for each slot, and shuffles validators to committees, in a way that each committee has at least 128 validators.

Validators can create attestations that stand up for the validity of blocks. The votes, proportional to the validators' balance, are recorded on the beacon chain. Votes attest that the head of the beacon chain is the block at slot X, and are called LMD GHOST votes. If a proposer receives enough votes from other validators (attesters), namely a supermajority (more than $\frac{2}{3}$ of the total validator stake), the block is considered justified, and the proposer receives a reward. The entire validator set has around half a million validators [157] and is capped at 2^{19} validators [158]. The consensus algorithm also incentivizes validators to report other validators that make conflicting votes (for example voting yes and no) or propose multiple blocks in the same slot (called slashing). When the first two blocks from two consecutive epochs are justified, we say that the previous block is finalized.

This process implies that validators are relatively synced with each other. The Beacon chain provides a global clock used for this synchronization. Each slot is defined as 12 seconds, and an epoch has 32 slots (6.4 minutes). A block is added to the blockchain for every slot, although these can be empty. Validators then have the incentive to attest to receive awards, and do it timely and be online (otherwise, the rewards will be minimized). Dishonest validators are disincentivized by the slashing caused by honest validators that accuse them. Validators are executed by validator clients (execution layer) connected to a beacon chain node (consensus layer). Consensus is done on the transactions to be recorded on the ledger. Transactions are included in blocks containing a block header and a list of transactions. Block headers in Ethereum contain several attributes, e.g.[24], previous header finalized slot, next header slot, signature slot, sync committee period slot signature, sync committee finalized header slot, previous header finalized state root, previous finalized state root branch, execution state root, and execution state root branch.

B. *Sync Committee*

In more detail, sync committees are composed of a fixed-size subset of the validators of the blockchain [159], [160], that are agreed upon by all honest validators. The next epoch committee is calculated in the current epoch. This way, the current sync committee can sign attestations about the following epoch (including the set of validator public keys that will form the next committee). Ideally, the sampled sync committee should retain an honest majority in each epoch: sampling depends on the implementation, but it could be done by sampling uniformly at random from the underlying stake distribution. The sync committee members receive a reward of around 0.1 Eth for every set of attestations in a sync period.

Light clients rely on block headers to make decisions without processing the entire blockchain. There are two types of headers: attested headers provide light clients with the most recent information (still not finalized), while the finalized header gives them a secure point of reference known to be irreversible (finalized). The attested header represents a beacon block that has reached finality, meaning it has passed the conditions set by the consensus algorithm to be considered irreversible.

APPENDIX D
A GENTLE INTRODUCTION TO SNARKS

Formally, SNARKS are cryptographic tools that enable a prover $\mathcal{P}$ to convince a (computational weak) verifier $\mathcal{V}$ of statements[25] of the form "given a function f that can be reduced to an efficiently computable boolean circuit $\mathcal{C}$ and an input $x_\mathcal{C}$, there exists a private witness $w_\mathcal{C}$ such that $f(x_\mathcal{C}, w_\mathcal{C}) = 1$. We call $(x_\mathcal{C}, w_\mathcal{C})$ an instance-witness pair. We define the index relation $R_\mathcal{C} = \{x, w\}$. The relation is an interactive protocol between a prover and a verifier such that the former can convince the latter that it knows a witness such that $(x, w) \in R$. The prover does so by sending polynomial oracles. The verifier can query these points of their choice. The more points queried and verified, the higher the confidence of the verifier that the prover indeed holds a valid witness belonging to R. A SNARK consists of a triple of probabilistic polynomial time algorithms (G, P, V) as follows [48], [162]:

- G is a generator that upon receiving a security parameter input λ, generates a reference string σ and a verification state τ.
- $P(\mathcal{C}, x_\mathcal{C}, w_\mathcal{C})$ that outputs a proof π for input $x_\mathcal{C}$ and witness $w_\mathcal{C}$ on circuit $\mathcal{C}$

---

[24] see the block header definition here.

[25] for a high-level introduction, see [161].

- V, is the verifier. It takes the verification state, instance, and proof and returns 1 if the proof is valid and 0 otherwise, i.e., $V(\tau, x_C, \pi) \rightarrow 1$ if a proof is valid.

SNARKs have the following properties:

- Correctness: A valid proof π is always accepted by an honest verifier V.
- Soundness: A proof π generated with an invalid witness (i.e., invalid proof) will not be accepted by an honest verifier with a high probability.
- succinctness: the cryptographic proof is small (few kilobytes) and easy to verify (few milliseconds, verifiable in polynomial time), regardless of the complexity of f. This proves useful for our specific real-world application.

In addition, zero-knowledge SNARKs (zkSNARKs) are a subset of SNARKS with the additional property:

- zero-knowledge property, which states that the interaction between the prover and the verifier does not reveal information about the witness.

Proving computation (i.e., creating a SNARK) comes with considerable overhead (in terms of time and memory) because the prover needs to express f as a Boolean circuit C that is much larger than the description of f, and the prover must perform expensive operations (time and memory) that grow at least in the size of C.

The pipeline from ideation to SNARK verification has several steps. In the first place, an idea is implemented as a program. However, typical high-level programming languages can not deal with SNARKs , and hence several domain-specific languages were developed. These domain-specific languages compile a program description into an R1CS. Then the circuit is fed into a proof system algorithm, that generates public parameters. Those parameters can be used to create proofs and later, do verification. One of the popular DSLs used for generating R1CS is circom, a hardware description language (meaning it is used to describe the arithmetic circuit directly).

1) Trusted Setup: Before SNARK is constructed and evaluated, a setup phase can exist (depending on the specific protocol). The setup phase improves the efficiency of the verification phase, by summarizing a circuit and outputting public parameters (common reference string, aka CRS). The circuit verifier uses the CRS to check for proof validity. The setup takes as input a set of parameters and a common reference string (CRS), which are r random bits, that must be kept secret from the prover and destroyer after the setup (otherwise, false statements can be proven). Groth16 [47], the SNARK proof construction protocol we use in this paper, uses a trusted setup.

2) Generating SNARKS: Although it is out of our scope for us to provide an in-depth explanation how SNARKS are constructed, we specify it informally: According to [163], a SNARK is obtained from a polynomial interactive oracle proof (PIOP) [164] and polynomial commitment schemes (PC schemes) [165]. In a PIOP, the verifier asks the prover to open all commitments at various points of their choosing using polynomial commitment scheme. The committment scheme allows a party to commit to a polynomial (typically a string) that can be used a posteriori by a verifier to confirm claimed evaluations of the committed polynomial. First, an interactive argument is constructed. The prover P invokes the prover algorithm $\mathcal{P}$ and the verifier invokes the verifier algorithm $\mathcal{V}$. P commits the polynomial oracles output using a polynomial committment and sends the results to $\mathcal{V}$. Verifier algorithm $\mathcal{V}$ declares the queries to the committed polynomials and then $\mathcal{P}$ replies with the polyonomials evaluations (a set of points), along with an evaluation proof (simplified). To turn this interactive protocol into a non-interactive protocol (suitable for our use case), one can use the Fiat-Shamir transform [166]. This works by having the prover generate randomness (used for the hidden challenge) on behalf of the verifier using a cryptographic hash function (random oracle model). The prover can then send a single message along with the randomness to the verifier, who then verifies the proof.

3) Verifying SNARKS: The evaluation process depends heavily on the protocol and implementation [26] (e.g., Groth16 [47], PLONK [164]), its implementation, and underlying cryptographic primitives (bilinear pairings, elliptic curve). For Groth16, it is roughly the following: Consider consider the common reference string (CRS) and the proof from the prover as (a, b, c), where a, b, and c are points on an elliptic curve. The public inputs are denoted by x. The verifier does the following:

1) Compute $h(x)$ using the public inputs x. This is done by evaluating the polynomial h at x, where h is the polynomial in the Quadratic Arithmetic Program that represents the public inputs.
2) The verifier checks a bilinear pairing equation such that the proof (a, b, c) satisfies the Quadratic Arithmetic Program represented by the CRS and the public inputs x [27].
3) The verifier accepts the proof if and only if the bilinear equation holds true.

4) Groth16: In this paper, we use the Groth16 construction to create SNARKS. Introduced by Jens Groth in 2016, the Groth16 SNARK construction stands as an efficient instantiation of zero-knowledge SNARK protocols, due to using elliptic curve pairings. The generated proofs are around 192 bytes. Groth16 uses a trusted setup phase.

APPENDIX E
EVALUATION PLOTS

Here, we showcase some plots supporting Experiment 2 from Section VI.

A theoretically safe approach for outdated devices to increase their safe syncing range requires a greater sync committee majority, i.e., greater than $\frac{2}{3}$. Such a greater majority would require more validators to turn malicious before the sync committee can be corrupted, thus increasing the required

---

[26] There are diverse tools and libraries allowing to verify SNARKS on-chain, most notably snarkjs [74].

[27] we omit technical details that can be consulted here, for example.

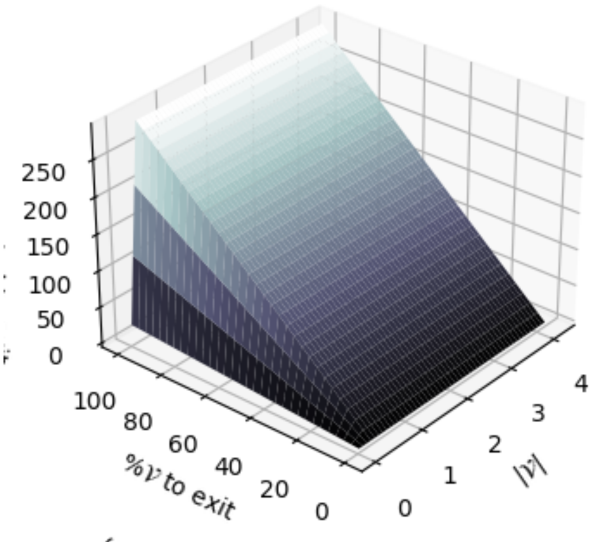


Fig. 14: Days needed for a fraction r of validators to exit Ethereum as a function of $|\mathcal{V}|$

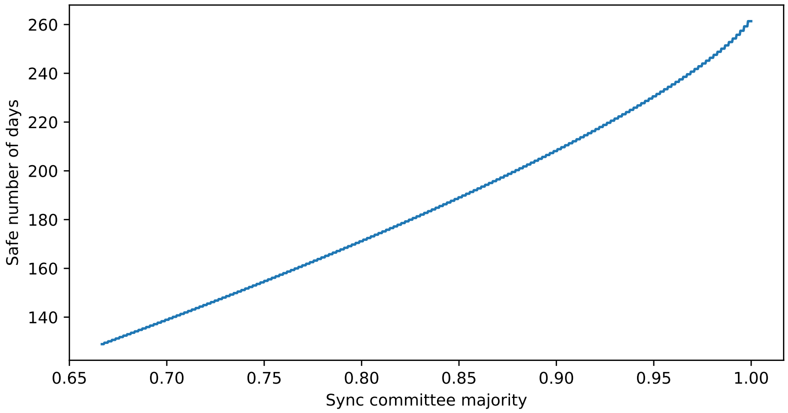


Fig. 15: Number of days for a malicious fraction of the validators to exit the system as a function of the honest sync committee ratio r

time for the sufficient number of validators to have exited. This means that in practice if the light client is presented with a chain of light client updates with higher sync committee participation, it can adjust its accepted syncing range accordingly. Let us s say a light client is outdated by 230 days. For the next 22 days of catching up it might require blocks, signed by at least 95% of respective sync committees. When the client is just 208 days behind, it might relax that requirement to just 90%. That dynamic adjustment keeps on until the client is ¡128.93 days behind when it falls back to the standard $\frac{2}{3}$. An analysis of all light client updates from the Altair fork (at slot 2375680) up to slot 4731420 shows that the minimal sync committee participation so far is 97%. Therefore, it would be safe for light clients to sync at least once every 230.5 days.

APPENDIX F
ALTAIR FORMAL SPECIFICATION

Figure 16 shows the definition of the `ValidateUpdate` algorithm for Altair.

APPENDIX G
CIRCOM TEST RESULTS

Table III shows the results of running our Circom tests with snarkit2.

APPENDIX H
DETAILED FUTURE WORK DIRECTIONS

This section illustrates future work and open challenges for SNARK-based bridges. We showcase use cases that can be built with Harmonia. We identify multiple trends:

Algorithm 7: ALC `ValidUpdate`

```
Input: $\mathcal{L}_{\mathcal{S}_i}$, $\texttt{LCU}_{data}, s, r$
Data: Access to blockchain $\mathcal{B}$
Result: true if validated header, otherwise $\perp$
1  assert $\sum \texttt{LCU}_{data}.aggregate \geq 1$
2  assert $s \geq \texttt{LCU}_{data}.signature\_slot > \texttt{LCU}_{data}.\texttt{BlockHeader}.slot \geq \texttt{LCU}_{data}.\texttt{F-BlockHeader}.slot$
3  $store_p = \texttt{SlotToPeriod}(\texttt{LCU}_{data}.\texttt{BlockHeader}.slot)$
4  $sig_p = \texttt{SlotToPeriod}(\texttt{LCU}_{data}.signature\_slot$
5  $sig_p = store_p$ ▷ if there is $\mathcal{L}_{\mathcal{S}_{i+1}}$, assert $sig_p = store_p + 1$
6  ▷ verify that the updated attested slot is higher than the one stored in the
     finalized header or that there is not a defined $\mathcal{L}_{\mathcal{S}_{i+1}}$
7  $\mathcal{B}$.state.verify($\texttt{LCU}_{data}.\pi_{\texttt{F-BlockHeader}_{i+1}}$, $\texttt{LCU}_{data}.\texttt{BlockHeader}.root$, $\texttt{LCU}_{data}$.F-BlockHeader.root)
     ▷ verify $\pi_{\mathcal{C}_{i+1}}$ authenticates F-BlockHeader
8  $\mathcal{B}$.state.verify($\texttt{LCU}_{data}.\pi_{\mathcal{C}_{i+1}}$, $\texttt{LCU}_{data}.\texttt{BlockHeader}.root$, $\texttt{LCU}_{data}.\mathcal{C}_{i+1}$) ▷ verify
     $\mathcal{C}_{i+1} = \texttt{BlockHeader}.\mathcal{C}_{i+1}$
9  $\mathcal{C}_i = \mathcal{L}_{\mathcal{C}_i}$ ▷ if update signature period = store period
10 $pubkey \leftarrow \texttt{AggregateToPubKey}(\mathcal{L}.aggregate)$
11 ▷ calculates $d = domain$
12 $m \leftarrow \texttt{ComputeRoot}(\texttt{LCU}_{data}.\texttt{BlockHeader}, d)$
13 $\sigma \leftarrow \mathcal{L}.aggregate.\sigma_{N_{[1:512]}}$
14 assert$\texttt{VERIFY}_{pubkey}(m, \sigma)$
15 return true
```

Description

In the Light Client Update `LCU` (Algorithm 7), we take as input the current state of the light client $\mathcal{L}_{\mathcal{S}_i}$, the light client update data $\texttt{LCU}_d$ata, a slot s and a root r. It returns true upon success. The procedure is as follows:

- In line 1 we check that the sync committee has sufficient participants
- In line 2, we verify the update does not skip a sync committee period
- Lines 3 and 4 calculate the updated signature and store periods from their slots
- Line 7 authenticates the finalized header via a Merkle proof
- Line 8 authenticates the next sync committee via a Merkle proof
- Line 10 calculates the public key that will verify the aggregate signature in line 14.
- We return true if all checks pass (line 15). If any check fails, we return $\perp$.

Fig. 16: `ValidateUpdate` in ALC

- *Continuous security improvements*: engineering efforts and implementation complexity bring about risks that are often unexpected. Even if the light client protocol specification and implementation are secure, and the underlying blockchains are secure, the SNARK frameworks and applications built on top of the light client still may have protocol and implementation bugs [167]. Using novel technologies and a complex codebase is an aggravating factor that can introduce attack vectors. Therefore, good practices and risk assessment in cross-chain technologies should be done for every cross-chain use case (see the good example of Uniswap [113]). In the unlikely scenario that the light client is compromised, the latency period we defined provides reaction time for various circuit breakers to be activated (e.g., emergency stop [12]). This would stop the cross-chain application and allow cybersecurity professionals to do the due diligence. In fact, systems like Hephaestus are fundamental to prevent propagation of safety failures (challenge $\mathcal{C}_2$).
- *Bridge aggregator rising popularity*: As the cross-chain market matures, bridge aggregator companies emerge, trying to build dependable and efficient swap routes on top of bridges. Bridge aggregators attempt to inherit the strengths of individual bridges and limit their weaknesses by tuning a balance between safety and liveness and each security model and assumption. Aggregators do a market study on bridges that meet a security threshold, for example, following the frameworks [2], [113]. Recently, Uniswap has shown interest in closely examining bridge aggregator development [168].
- *Standardization efforts catching momentum*: diverse standards are being developed with applications on permissioned and permissionless blockchains. An example that links the permission and permissionless worlds is IETF's SATP [7], [91], counting with the support of companies like Blockdaemon, Quant, and diverse research institu-

	Template Instances	Non-Linear Constraints	Linear Constraints	Public Inputs	Public Outputs	Private Inputs	Private Outputs	Wires	Labels
light_client	410	89,687,265	5,093,393	0	2	20,961	0	93,965,192	472,858,554
aggregate_bitmask_N1	48	11,664	0	0	14	15	0	11,592	13,509
aggregate_bitmask_N3	50	35,452	0	0	14	45	0	35,226	40,619
compress	9	809	2	0	384	14	0	812	1323
compute_domain	101	58,854	0	0	256	320	0	58,551	411,153
compute_signing_root	101	59,281	0	0	256	512	0	59,170	411,089
division_by	6	757	0	0	2	2	0	758	1279
expand_message	102	559,346	0	0	2048	256	0	553,668	3,884,336
hash_to_field	121	573,506	0	0	28	256	0	567,632	3,904,808
hash_tree_root	101	177,843	0	0	256	1024	0	176,996	1,231,473
hash_tree_root_beacon_header	102	413,348	0	0	256	1,280	0	410,261	2,875,313
hash_two	100	59,281	0	0	256	512	0	59,170	410,065
is_equal_arrays	3	8	0	0	1	6	0	15	35
is_first	4	5	0	0	1	4	0	10	24
is_supermajority	4	252	0	0	0	1000	0	1,252	1,262
is_valid_merkle_branch	106	299,493	0	0	0	1,793	0	298,166	2,056,242
less_than_bits_check	4	97	0	0	1	2	0	97	171
less_than_eq_bits_check	5	97	0	0	1	2	0	97	174
numbersTo256Bits	3	256	0	0	256	2	0	257	773
pow	4	1021	0	0	1	2	0	1,024	2,311
range_check	5	195	0	0	1	3	0	193	347
selector	3	24	0	0	1	9	0	34	68
ssz_num	2	32	224	0	256	1	0	257	323
sync_commitee_hash_tree_root	103	531,769	0	0	256	1,920	0	528,074	3,688,785

TABLE III: Circom test results

tions. In the permissionless world, a relevant standard is MMA, a standard for building a quorum of approvals from different bridges that propagates to a smart contract on a target blockchain. Nonetheless, they currently focus on token transfers, finding an optimal route between chains (i.e., the cheapest bridge that can transfer token X from A to B). This is not an easy challenge, as the routing algorithms combine gas estimation, slippage, liquidity pool monitoring, and synchronizing different blockchains. Recent research showcases an aggregator of bridge aggregators [169]. LiFi is one of the companies implementing MMA. The development of technologies like Harmonia can feed back on standards development, and vice-versa. Moreover, ecosystem-dependent standards are also being developed, for example for IBC [170], [171] and EVM [172]

- *Privacy-preserving bridges*: in this work, we presented Harmonia, which leaks all information to the adversary. While achieving desired security properties, one might want to provide privacy, as it is increasingly more important [173]. Privacy-preserving applications built on top of Harmonia, such as bridges, could eventually be implemented [135], but the correct privacy model is still not clear: what are the ideal functionalities (in terms of the UC framework [174]) that can guarantee that different privacy models co-exist? For instance, can asset transfers between a private and a public blockchain not leak any information according to a set of defined privacy policies? What are the functionalities of each combination blockchain pair-bridge that need to be assured?
- *Cross-chain monitoring*: there is a large literature on blockchain security [175], and more recently, some works appeared on the security of cross-chain technologies. Although a few initial monitoring tools look promising to the field [12], [176], applicability and efficacy in real-world products such as Chainalysis still need to be assessed. Furthermore, large-scale empirical studies on cross-chain attacks do not exist.
- More efficient on-chain proofs: to prove state from the Ethereum blockchain, to be used by third-party chains, we provided a SNARK attesting for the correct execution of the light client rules. This SNARK will originate block roots that can be consumed by Merkle proof verifiers to verify Merkle proofs. There could be a more efficient so-

lution. In the future, where volume is substantially higher, we could use a SNARK to aggregate a chain of Merkle proofs, instead of providing a SNARK proving that a light client protocol update was correctly executed. This way, the bridge consumes SNARKs to validate cross-chain logic and will not depend on validated block headers. More research is needed to understand when starting to compress Merkle proofs as a SNARK versus using a SNARK per a range of blocks will be economically viable.