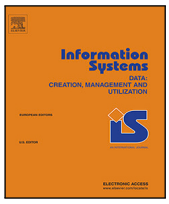
Information Systems 141 (2026) 102764

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



zkPACT: A zero-knowledge private cross-chain token transfer framework utilizing decentralized oracle networks☆

Elmira Ebrahimi [a], Anh-Tu Hoang [a], Dominik Kaaser [a], Michael Sober [a], Juan M. Tirado [b], Stefan Schulte [c,*]

[a] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Institute for Data Engineering, TU Hamburg, Hamburg, Germany
[b] Bitpanda GmbH, Vienna, Austria
[c] Christian Doppler Laboratory for Blockchain Technologies for the Internet of Things, Software and Business Engineering, TU Berlin, Berlin, Germany

ARTICLE INFO

Keywords:
Blockchain interoperability
Decentralized oracles
Cross-chain token transfers
Privacy-preserving protocols
Zero-knowledge proofs
Blockchain-based information systems

ABSTRACT

Despite the growing adoption of blockchains, their isolated architectures hinder seamless cross-chain communication, challenging applications that rely on integrated blockchain infrastructures, notably Blockchain-based Information Systems (BISs). Achieving interoperability while preserving privacy and regulatory compliance remains a core challenge, particularly when separate organizations operate different blockchain platforms and tokenized value must move across them without exposing transaction links that may reveal business relationships or payment behavior. Existing interoperability solutions often incur high computational overhead and rely on protocol-specific assumptions, limiting their applicability across heterogeneous blockchains.

We introduce zkPACT, a privacy-preserving framework for compliant cross-chain token transfers across heterogeneous blockchains. Our framework combines Zero-Knowledge Proofs (ZKPs), oracle networks, and off-chain batching to support scalable transfers. It employs a coordinated oracle model in which validators process cross-chain burn events, while a rotating aggregator updates the shared off-chain Merkle tree after reaching consensus, enabling private and efficient token claims. To improve scalability and reduce gas costs, zkPACT batches claim requests off-chain and then submits a single succinct proof to the smart contract. To ensure validator accountability, the framework enforces an incentive mechanism and dynamic slashing. We also integrate a Know Your Customer (KYC) mechanism that enables users to demonstrate compliance without revealing sensitive data, preserving privacy and accountability in the event of abuse. We present a proof-of-concept implementation of zkPACT that achieves up to 95% lower gas costs and up to 94% lower off-chain memory usage than a non-batching approach, demonstrating its suitability for private, scalable cross-chain token transfers.

1. Introduction

Blockchains operate as decentralized ledgers, ensuring the immutability of transaction records while enhancing transparency and efficiency [1]. One prominent application of blockchain technology is Blockchain-based Information Systems (BISs), using blockchain's core properties to support decentralized trust and auditable management of digital assets and data within information system architectures [2]. BISs are already used in domains such as healthcare [3], supply chain management [4], and Internet of Things (IoT) [5], where trustworthy record-keeping and verifiable data sharing are essential.

In the current blockchain landscape, many networks operate under different consensus mechanisms, token standards, and governance models [6]. As a result, applications built on separate ledgers cannot easily share data or transfer assets across networks, limiting cross-organizational interaction.

To address these challenges, interoperability mechanisms are required [7], enabling secure and efficient cross-chain communication for BISs and other blockchain applications deployed across heterogeneous blockchains. Blockchain interoperability approaches range from earlier mechanisms, such as Hashed Time-Locked Contracts (HTLCs), notary schemes, sidechains, and relay-based systems [8], to broader architectures, such as blockchain-of-blockchains, DApp-based connectors, and gateway-based mechanisms [9]. Earlier mechanisms support atomic exchange, delegated validation, auxiliary-chain coordination, or remote-state verification, while broader architectures use shared infrastructure,

☆ This article is part of a Special issue entitled: 'BISs' published in Information Systems.

* Corresponding author.
E-mail address: schulte@tu-berlin.de (S. Schulte).

https://doi.org/10.1016/j.is.2026.102764
Received 27 September 2025; Received in revised form 8 May 2026; Accepted 28 May 2026
Available online 11 June 2026

connector layers, or gateway services to expand cross-chain interaction. Despite this progress, existing approaches still involve trade-offs in trust assumptions, coordination overhead, platform-specific integration, and verification complexity, often leading to high on-chain computational overhead and gas costs [10].

Recent studies [11,12] have explored the use of ZKPs to improve blockchain interoperability by shifting intensive computation off-chain, enabling efficient on-chain verification and reducing gas costs. Despite these advances, current solutions still fall short in three aspects. First, most ZKP-based interoperability frameworks focus on proving the correctness of cross-chain messaging or state verification, but do not provide transaction unlinkability for value transfer across chains, so cross-chain activity remains correlatable [10]. Second, some approaches (e.g., [13]) generate and verify a separate ZKP for each cross-chain request, increasing computational overhead and gas consumption. Third, some privacy-preserving designs (e.g., [14]) omit compliance requirements or handle them outside the protocol rather than through protocol-verifiable eligibility and revocation proofs during operation. This separation can complicate cross-organizational deployment and increase the risk of identity disclosure [15]. These limitations are important for BISs, where cross-platform value transfer should remain unlinkable to avoid exposing business relationships or asset positions while satisfying institutional onboarding requirements.

To this end, we introduce zkPACT, a privacy-preserving framework for cross-chain token transfers using decentralized oracles [16] and Zero-Knowledge Succinct Non-interactive Arguments of Knowledge (zk-SNARKs) [17]. In zkPACT, oracle nodes serve dual roles as validators and off-chain mixers [18]. They verify burn events across multiple blockchains and pool burn commitments into a shared Merkle tree, thereby preserving transaction unlinkability. To improve scalability, our framework processes token claim requests in off-chain batches. Validator votes are aggregated into a single ZKP, which is verified on-chain to update the system state. This zk-rollup-style design eliminates repeated on-chain checks and reduces gas costs by verifying one proof per batch rather than one per claim [19]. The framework also employs incentive and slashing mechanisms to ensure integrity.

To support privacy-preserving transfers in regulated cross-organizational settings, zkPACT further integrates a Self-Sovereign Identity (SSI)-based Know Your Customer (KYC) mechanism [20], enabling compliance checks without disclosing full identity information while preserving accountability in misuse cases. Building on this foundation, our key contributions are as follows:

- We present a privacy-preserving cross-chain token transfer framework that ensures transaction unlinkability and confidentiality via ZKPs and decentralized oracle nodes that serve dual roles as validators and off-chain mixers.
- To enhance throughput and reduce on-chain costs, we use zk-rollups to offload computations and a batching technique to verify multiple claim requests with a single ZKP.
- Our framework introduces a dynamic incentive model that combines stake-weighted reputation and slashing to promote honest behavior and penalize misbehavior.
- We ensure regulatory compliance by integrating a KYC mechanism, balancing privacy with accountability under Financial Action Task Force (FATF) standards [21].

The remainder of this paper is structured as follows. In Section 2, we review the foundations of blockchain interoperability and key underlying concepts. We review related work and highlight the challenges associated with privacy-preserving cross-chain solutions in Section 3. We present the design of our framework in Section 4 and describe the implementation details in Section 5. We evaluate our system's computational performance in Section 6 and discuss its security and privacy in Section 7. Finally, in Section 8, we summarize key findings and discuss future research directions.

2. Background

2.1. Blockchain interoperability

Blockchain interoperability refers to the ability of blockchain networks to communicate and exchange data seamlessly across different blockchains [6]. This addresses inherent limitations of blockchain networks, which are designed as independent systems with specific protocols, consensus mechanisms, and governance structures. By enabling blockchains to communicate, interoperability mechanisms [7] support the development of blockchain-driven applications, such as BISs [1]. These mechanisms enable three key functionalities [22], namely data transfer, asset exchange, and asset transfer.

Data transfers [23] enable blockchains to exchange information such as transaction details and smart contract states. Asset exchanges [24] allow assets to be swapped between blockchains while remaining on their respective networks. In contrast, asset transfers [25] move assets from one blockchain to another while preserving their value, typically by locking or burning them on the source chain and minting an equivalent amount on the target chain. Some implementations synchronize the source blockchain's consensus mechanism with the target chain [26], while others use ZKPs [27] for privacy and efficiency.

These interoperability functionalities can be implemented through architectures such as sidechains [7], blockchain-of-blockchains [28], relays [29], and notary schemes [10], each involving trade-offs in decentralization, trust, and performance. Among these approaches, notary schemes leverage a trusted entity to monitor blockchains and facilitate cross-network transactions, providing a flexible solution for cross-chain communication. Notaries can operate independently of the underlying consensus or architectures of the blockchains involved, making them easy to deploy across heterogeneous environments. However, traditional notary schemes rely on a centralized trust model, introducing concerns about censorship resistance and single points of failure.

To overcome these limitations, recent studies [30] employ committee-based notary models in which decentralized nodes collaboratively relay cross-chain messages. These designs enhance fault tolerance and reduce reliance on single entities. Decentralized oracle networks [16] extend this idea by acting as distributed notaries between blockchains. Oracle nodes serve as bridges between blockchains and external data sources for secure data transmission. Unlike centralized oracles, these networks employ validator committees to collectively retrieve, verify, and submit information across chains.

While interoperability frameworks enable cross-chain communication, scalability is limited by high on-chain costs. Layer-2 solutions such as rollups [31] offload transaction execution and reduce on-chain computation, thereby addressing scalability limitations. Optimistic rollups [32] assume transaction validity and rely on a dispute window during which incorrect results can be challenged using fraud proofs. This improves scalability but introduces latency due to the challenge period. In contrast, zk-rollups [19] generate succinct ZKPs (see Section 2.2) to attest to the correctness of off-chain execution, enabling faster finality and stronger security guarantees.

2.2. Zero-knowledge proofs

ZKPs [33] are cryptographic protocols which allow a prover to convince a verifier that a witness is correct without revealing any additional information beyond its validity. Formally, ZKP systems operate over a relation R defined on tuples of public statements x and corresponding private witnesses w, where $(x, w) \in R$ implies that w is a valid witness for x. The system relies on a proving key pk, used by the prover to generate proofs, and a verification key vk, used by the verifier to validate proofs. A ZKP system must satisfy three fundamental properties to ensure security and reliability [34]:

1. Completeness. If the prover possesses a valid witness w for statement x, an honest verifier will always accept the proof.

2. Soundness. A dishonest prover cannot convince an honest verifier of a false statement. This guarantees that an adversary cannot generate a valid proof of an incorrect assertion.

3. Zero-Knowledge. The proof should reveal no information about the witness w beyond the validity of the statement, thereby ensuring privacy while maintaining verifiability.

Among the various ZKP protocols, zk-SNARKs [17] are widely used in blockchain systems [35] due to their small proof size and constant verification time. One of the most efficient zk-SNARK constructions is the Groth16 proving scheme [36], which optimizes proof size and verification complexity by leveraging elliptic curve pairings and polynomial commitments. The Groth16 protocol encodes computational problems as polynomial equations over a finite field using arithmetic circuits. Such circuits are directed acyclic graphs with input nodes representing variables or field elements, gates performing basic arithmetic operations, and output nodes producing results, all connected by edges that define the flow of computation.

The computation to be proven is first expressed as an arithmetic circuit that models the program's logical steps. This circuit is then transformed into a Rank-1 Constraint System (R1CS), defining constraints that ensure the computation was carried out correctly. To enable efficient verification, the R1CS is compiled into a Quadratic Arithmetic Program (QAP), expressing the constraints as polynomial identities over a finite field. This QAP-based representation forms the basis of the Groth16 protocol, which operates in three main phases:

1. Setup. The program logic is compiled into an arithmetic circuit and encoded as a constraint system. A Common Reference String (CRS) is generated, where the proving and verification keys are computed as $(\text{pk}, \text{vk}) \leftarrow \text{Setup}(C, \lambda)$ based on a computational circuit C and a security parameter λ. To ensure trust, the CRS should be securely generated, for example, using Multi-Party Computation (MPC) to distribute the setup [37].

2. Proof generation. Using the proving key pk, the prover computes a succinct proof as $\pi \leftarrow \text{Prove}(\text{pk}, x, w)$ that attests to the correctness of a computation for a public statement x and a private witness w. The proof satisfies a pairing relation, enabling efficient verification without revealing w.

3. Verification. The verifier uses the verification key vk to check whether the proof π is valid for the public input x by verifying that $\text{Verify}(\text{vk}, x, \pi) = 1$, which ensures x satisfies the computation's constraints.

2.3. Self-sovereign identity

SSI is a decentralized identity management approach that allows users to control their digital identities, avoiding repeated verification by centralized authorities [20]. A SSI framework typically consists of three roles, namely issuer, holder, and verifier. Issuers are trusted entities, such as government agencies or regulated financial institutions, that issue identity credentials to users after verifying their real-world identity. Users, referred to as holders, store these credentials locally. Verifiers are external parties who define specific conditions for granting access to services and validate user credentials against those conditions. Digital credentials in SSI are expressed as Verifiable Credentials (VCs), which are cryptographically signed statements linking a set of attributes, such as nationality, residency, or age, to a user. A holder can selectively prove these attributes to a verifier using a Verifiable Presentation (VP), which is a derived proof that supports fine-grained disclosure while preserving the authenticity of the underlying credential.

To support scalable, privacy-preserving revocation, each issuer maintains a revocation tree, assigning each credential a unique index [38]. Revoked credentials occupy their corresponding leaves, while non-revoked credentials correspond to empty leaves. To prove validity, the holder provides a Proof-of-Non-Revocation (PoNR), i.e., a Merkle path from the expected empty leaf to the root. The verifier then checks policy compliance, non-revocation, and trusted issuance without learning unnecessary information about the user or other credentials.

3. Related work

3.1. Cross-chain mechanisms

Several architectural paradigms have been proposed to enable blockchain interoperability. Huang et al. [39] propose an HTLC-based cross-chain asset transfer protocol that introduces trusted middlemen, along with anonymous identity authentication and trust evaluation, to improve the reliability of cross-chain execution. Similarly, HT2REP [40] is a protocol with strong composable security guarantees that employs time-released encryption to prevent fairness attacks. While HTLCs are widely adopted to ensure atomicity and minimize trust assumptions, they rely on time-locked scripts that expose swap conditions on-chain, require synchronous communication, and struggle to scale in high-throughput environments.

Pathak et al. [41] propose SATI, a trust and access control mechanism for secure data sharing in IoT environments that uses a sidechain architecture to enable cross-chain transfers. It combines local trust scoring with cross-chain synchronization of access policies between a sidechain and a main blockchain. Although SATI enhances scalability and coordination, it lacks cryptographic techniques to protect transaction or access pattern privacy, leaving it open to inference attacks. Generally, sidechains rely on validator sets to maintain peg security, introducing centralization risks and governance overhead.

Cao et al. [29] propose a relay-based design with on-chain light clients and optimized verification to reduce cross-chain verification overhead across heterogeneous blockchains. Guo et al. [42] propose xRWA, a framework for real-world asset transfer, combining Simplified Payment Verification (SPV)-based authentication with verifiable credentials for cross-chain identity recognition. However, although relay-based designs reduce trust assumptions by verifying remote chain state, maintaining the verification logic and client state requires continuous updates, increasing execution and storage overhead, which can be costly for resource-constrained chains [43].

Notary-based models offer adaptable and straightforward designs. Ren et al. [30] propose a method that combines a notary group with verifiable secret sharing to address timeout attacks and improve resilience. Sun et al. [44] introduce a hybrid notary-hash-locking protocol with incentive mechanisms to discourage misbehavior. While these models simplify cross-chain deployment, centralized notaries remain a concern. Recent advancements mitigate some of these issues through multi-signature validation and decentralized oracle networks [23].

Peelam et al. [28] analyze Cosmos as a blockchain-of-blockchains architecture, where application-specific zones interoperate through a central hub for cross-chain data and asset exchange. Morháč et al. [45] propose UniSpell, a universal adapter for the Polkadot ecosystem that exposes routing, transfer, and swap functionality to DApps through native cross-chain messaging. While these platform- and middleware-level approaches broaden interoperability beyond individual bridge protocols, they remain tied to ecosystem-specific assumptions, connector logic, or messaging infrastructures.

Shortcomings of Existing Cross-Chain Mechanisms. Despite improving interoperability across heterogeneous blockchains, these mechanisms still face two limitations. First, most of them expose key cross-chain transaction information during execution, which leaves transfer activity observable and often linkable across chains. Second, many of these designs impose considerable on-chain or coordination overhead, for example, through repeated verification steps, relay maintenance, or multi-party interaction, which reduces efficiency as the number of transfers grows. These limitations have motivated a growing line of work that incorporates ZKPs into cross-chain protocols.

Table 1
ZkPACT vs. representative ZKP-based cross-chain systems.

Protocols	Private transfer	Unlinkability	Batch finalization	Compliance	Accountability
zkPACT	✓	✓	✓	✓	✓
zkBridge [12]	✗	✗	○	✗	✗
Harmonia [27]	✗	✗	○	✗	○
zkOracle [23]	✗	✗	✗	✗	✓
Wu et al. [14]	✓	✓	✗	✗	✗
zkCross [11]	✓	✓	✗	✗	○
Chang et al. [46]	✓	✗	✗	✓	✗
Sober et al. [47]	✗	✗	✓	✗	○
Hu et al. [48]	✓	✓	✓	✗	✗

✓ supported; ✗ not supported; ○ partially supported.

3.2. ZKP-enhanced cross-chain mechanisms

Existing ZKP-based cross-chain protocols differ across several dimensions, including privacy, batching, compliance, and accountability, as summarized in Table 1. Here, *Private transfer* denotes protection of transfer-sensitive information, while *Unlinkability* denotes resistance to linking the source-side and target-side actions.

zkBridge [12] is a trustless cross-chain bridge that enables a receiver chain to verify the sender chain's state via a ZKP. Its batching reduces the cost of maintaining the sender-chain view by providing consecutive block headers in batches, but this batching is limited to header synchronization rather than user transfers. The design lacks support for private transfers, unlinkability, and protocol-level compliance proofs, and leaves relay-node accountability unspecified [12].

Harmonia [27] is a ZKP-based light-client interoperability framework in which destination chains verify source-chain state through ZKP-assisted light-client updates. It supports cross-chain use cases such as asset transfer and state migration without relying on a trusted operator. The framework partially supports batching by amortizing light-client update processing work, and supports accountability by discussing slashing for misbehaving parties. However, Harmonia does not target private transfers, burn-claim unlinkability, or protocol-level compliance proofs as design goals.

Sober et al. [23] propose zkOracle, an oracle-based system for cross-chain data transfer in which oracle nodes validate messages and submit proofs on-chain. Since each request is validated and proved separately, the design incurs high off-chain overhead and gas costs. It also lacks privacy-preserving mechanisms, leaving transactions fully traceable, and does not support protocol-level compliance proofs.

Wu et al. [14] propose a notary-based private cross-chain transfer scheme that hides sender, recipient, and amount using ZKPs, and traces malicious notaries via traceable group signatures. The protocol relies on multiple proof-generation steps rather than batch finalization with a single proof verification, and its trust model excludes protocol-level compliance proofs and validator accountability for claim processing.

zkCross [11] proposes a two-layer architecture for privacy-preserving cross-chain operations, consisting of a dedicated audit chain together with protocols for transfer and exchange. It supports private transfers and unlinkability by hiding the receiver's identity with zk-SNARKs and concealing amounts via fixed denominations. Its aggregation improves efficiency on the audit chain, but transfers still rely on per-transfer proof generation rather than batch finalization with a single proof verification. The framework further lacks protocol-level compliance proofs and relies on committer and auditor roles with rewards and an honest-committer assumption, without a slashing-based accountability mechanism for misbehavior.

Chang et al. [46] propose a privacy-preserving KYC verification framework for interoperable payments, where users satisfy compliance checks without revealing identity attributes. The workflow verifies presented proofs and payment certificates along the payment path and maintains an auditable transaction history for settlement and accountability. The framework supports private transfers and compliance, but not unlinkability across payments, batch finalization with a single proof verification, or an incentive-based accountability mechanism.

Recent studies adopt batching to reduce on-chain verification costs. Sober et al. [47] present a zk-rollup-based cross-chain token transfer mechanism that shifts computation off-chain and posts only state commitments on-chain, but transactions within each batch remain linkable. The protocol mentions an incentive mechanism, yet does not provide protocol-level compliance proofs. Hu et al. [48] propose a privacy-preserving aggregated proof approach for batch cross-chain transactions, where multiple cross-chain blocks are grouped and verified with a single proof. The design focuses on verification efficiency and privacy, but does not address protocol-level compliance proofs or accountability mechanisms for participating roles.

Shortcomings of Existing ZKP-Based Mechanisms. Despite progress in ZKP-enhanced cross-chain systems, significant gaps remain in privacy-preserving cross-chain token transfers in BISs. Correctness-oriented bridges reduce trust in cross-chain verification but do not provide transfer unlinkability. Privacy-preserving designs improve confidentiality but typically rely on per-transfer proof generation and finalization, which ties overhead to the number of transfers. Batching-oriented approaches reduce verification overhead but do not integrate batching with transfer unlinkability and protocol-level compliance. For BISs, these limitations are especially restrictive as cross-platform transfer should remain unlinkable to avoid revealing sensitive transaction patterns, while participation must still satisfy eligibility requirements. These gaps motivate zkPACT, which we present next as a framework that combines cross-chain unlinkability, batched verification, and credential-based compliance.

4. Framework design

This section presents zkPACT, a framework for compliant, privacy-preserving cross-chain token transfers across heterogeneous blockchains. Designing such a framework in a decentralized oracle setting requires addressing key challenges. These include preserving privacy during committee-mediated cross-chain execution, supporting compliance without exposing sensitive user information, ensuring that oracle-side off-chain computations can be efficiently checked on-chain, and maintaining accountability despite asymmetric off-chain roles. We address these challenges by combining ZKPs, oracle nodes that act as off-chain validators and mixers, protocol-level eligibility and non-revocation proofs, batched claim finalization with a single on-chain ZKP verification per batch to reduce on-chain costs, and a dynamic incentive and slashing mechanism for validator accountability. We first present the system architecture, then describe KYC-based compliance, batch verification, and validator incentives, and finally explain the system workflow and the circuits that ensure off-chain computational correctness.

4.1. System overview

As shown in Fig. 1, zkPACT involves five key entities: a decentralized file system (DFS), an issuer, users, off-chain oracle nodes,

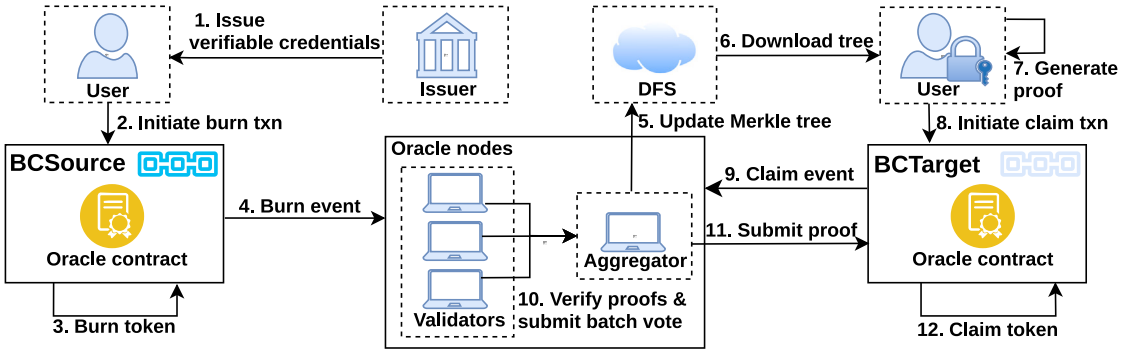


Fig. 1. Overview of the system.

and an oracle smart contract. The DFS maintains Merkle tree data off-chain, and the issuer handles identity verification. Users interact with the system both as initiators of burn transactions on the source blockchain (*BCSource*) and as claimants who initiate claim transactions on the target blockchain (*BCTarget*). The oracle nodes perform off-chain validation and coordination. The oracle contract is deployed on both blockchains and handles cross-chain enforcement by facilitating burn transactions, processing claim transactions, and verifying aggregated proofs to maintain transaction integrity. This separation of issuer, storage, validation, and settlement responsibilities keeps the interfaces between the participating components explicit and reduces system complexity by allowing each part of zkPACT to focus on a specialized function, which is consistent with the layered organization view of BISs [49]. Before describing the participating entities, we formally define the commitment hash and the burn and claim transactions. zkPACT processes transfers through denomination-specific contract instances, so all burns submitted to a given instance correspond to the same predefined amount. At a high level, the commitment hash serves as the privacy-preserving link between a burn on *BCSource* and a later claim from that pool. The burn transaction requests destruction of the predefined amount on *BCSource* and carries a commitment hash, which is later pooled by the oracle nodes into a shared commitment tree. The claim transaction then proves the right to redeem one commitment from the same pool on *BCTarget* without revealing which burn it corresponds to.

Definition 4.1 (*Commitment Hash*)**.** The commitment hash is defined as $c := h(nul \parallel sec \parallel dest_{id} \parallel VC_{hash})$.

The commitment hash includes the nullifier nul, whose hash nul_{hash} is revealed at claim time to bind the claim transaction to the commitment and prevent double-spending on *BCTarget*. The secret sec adds randomness for unlinkability across transactions, $dest_{id}$ binds the commitment to the intended *BCTarget* and prevents the same commitment from being used to claim tokens on multiple blockchains, thereby mitigating cross-chain double-spending. Additionally, VC_{hash} binds the claim to a VC for eligibility and accountability.

Definition 4.2 (*Burn Transaction*)**.** Let d denote the fixed token denomination supported by the protocol. A burn transaction on *BCSource* is defined as $\beta = c$, where c is the commitment hash later redeemed on *BCTarget*.

Definition 4.3 (*Claim Transaction*)**.** Let d denote the fixed token denomination supported by the protocol. A claim transaction on *BCTarget* is defined as $\gamma = (\pi, nul_{hash}, VC_{hash})$, where π is a ZKP attesting to the validity of the claim, nul_{hash} is the hash of the nullifier, and VC_{hash} is the hash of the submitted credential.

In cross-chain token transfers, each burn maps directly to a claim transaction. To prevent linkability, zkPACT integrates a mixing layer. A mixer [18] transforms incoming transactions into outputs such that no observer can determine which input produced which output. In zkPACT, each burn transaction is encoded as a commitment hash (see Definition 4.1) and inserted into a commitment tree representing the shared transaction pool. During the claim phase, users prove inclusion of their commitment in that tree without revealing the corresponding burn transaction. This provides anonymity within the pool and unlinkability between burn and claim transactions.

Maintaining the commitment tree on-chain would incur significant gas and storage costs. zkPACT therefore uses a DFS, where the blockchain stores only the latest root, while the DFS stores the commitment-tree leaves and the set of spent nul_{hash} values. This separation reduces on-chain overhead while providing users and validators with a consistent off-chain reference for proof generation and verification. However, mixing alone does not address misuse under unrestricted anonymity; zkPACT also integrates a KYC mechanism in which a trusted issuer verifies the user's real-world identity (Fig. 1, Step 1) and issues a VC attesting to eligibility (see Section 4.2).

After receiving the VC, the user computes the commitment hash c and submits the burn transaction β (see Definition 4.2) to the oracle contract on *BCSource* (Step 2). The oracle contract executes β and emits an event (Steps 3–4). Off-chain oracle nodes are essential intermediaries in our framework, playing a dual role as validators responsible for verifying token burn events and processing token claim proofs, and as mixers, ensuring transaction privacy by obfuscating transfer histories. Each *BCSource* has a committee of validators that listens to events from all involved blockchains. Upon detecting a burn event, validators update their local commitment trees for the mixing service and submit the result to the aggregator, which then updates the commitment tree in the DFS (Step 5).

During the token claim phase, the user retrieves the latest commitment-tree state from the DFS (Step 6) and generates a ZKP (Step 7). The user then submits a claim transaction γ (see Definition 4.3) on *BCTarget* (Step 8), after which the oracle contract emits a claim event (Step 9) to notify the validators. Validators verify the proof in a batch (see Section 4.3) and submit their votes to the aggregator (Step 10). The aggregator determines the consensus and generates a ZKP proving the correctness of vote aggregation, reward and penalty assignment (see Section 4.4), and validator-state updates, then submits the final proof to the oracle contract for verification (Step 11). The oracle contract on *BCTarget* mints tokens for the approved batch and updates the root of the validator-state tree (Step 12). This tree records the validator attributes required for accountability, with each validator assigned a fixed index whose leaf stores the hash of these attributes (see Section 4.4). We next present the core mechanisms of zkPACT before showing how they are integrated in Section 4.5.

4.2. KYC integration and accountability mechanisms

The following expands the KYC process from Step 1 in Fig. 1, explaining its operation and impact on subsequent steps.

A common concern in privacy-preserving BISs is that organizations need to impose both restricted participation and accountability in cases of illicit activity. zkPACT therefore aims to preserve protocol-level anonymity for honest users while enabling accountability for proven misbehavior. In particular, zkPACT aligns with financial compliance frameworks such as Anti-Money Laundering (AML) directives and Countering the Financing of Terrorism (CFT) measures, following guidance from FATF [21]. These frameworks require identity verification through KYC mechanisms, in which users present official documents to a regulated authority.

To meet compliance and accountability requirements, zkPACT integrates a KYC mechanism based on SSI principles (see Section 2.3), which follows the zkSSI framework [50]. It combines on-chain credential revocation with trusted issuers that maintain off-chain identity records, preventing users flagged for abuse from obtaining new credentials while preserving privacy during normal protocol operation. As a result, anonymous identity verification is achieved using ZKPs over the user's VCs and their corresponding validity conditions, rather than through a separate middleware layer.

zkPACT is designed to operate on top of existing KYC infrastructures instead of replacing them. The issuer is the regulated institution or compliance provider that performs document verification, while retaining customer audit data within its existing compliance environment. The additional integration required by zkPACT is limited to a credential layer that exports the outcome of that process in verifiable form. After successful onboarding, the issuer generates a signed VC containing attributes needed by zkPACT, publishes its public verification key, and maintains credential status through a revocation tree whose current root is made available to validators. To formalize how zkPACT expresses compliance requirements and how users prove them during claims, we next define the structure of the issued VC, the KYC policy, and a simple policy example.

Definition 4.4 (*VC*). Let u be a user and pk_{issuer} be the public key of a trusted issuer. The credential issued to u is denoted by $VC_u = \langle sub_u, pk_{issuer}, attrs_u, t_e, idx_u, sig_{issuer}\rangle$, where sub_u identifies the holder, $attrs_u$ contains the certified attributes used by zkPACT, t_e is the credential expiry epoch, idx_u is the credential's index in the issuer's revocation tree, and sig_{issuer} is the issuer's digital signature over the credential.

Definition 4.5 (*KYC Policy*). A KYC policy in zkPACT is represented as $\mathcal{P}_{KYC} = \langle C_{KYC}, \varphi_{KYC}\rangle$, where $C_{KYC} = \{c_1, \ldots, c_m\}$ is a finite set of credential checks and φ_{KYC} is a Boolean formula over them. Each check has the form $c_j = \langle attr_j, op_j, val_j, pk_{issuer,j}\rangle$, meaning that attribute $attr_j$, certified by issuer $pk_{issuer,j}$, must satisfy $attr_j\ op_j\ val_j$. A credential satisfies $\mathcal{P}_{KYC}$ if the checks in C_{KYC} evaluate to true under φ_{KYC}.

Example 4.1 (*Nationality-based Policy*). One such policy can be written as $\mathcal{P}_{KYC} = \langle \{c_{nat}, c_{exp}\}, c_{nat} \wedge c_{exp}\rangle$, where $c_{nat} = \langle \texttt{nationality}, \in, \mathcal{A}, pk_{issuer}\rangle$ and $c_{exp} = \langle \texttt{expiry\_epoch}, \geq, epoch_{claim}, pk_{issuer}\rangle$. The user later proves that the submitted credential satisfies $\mathcal{P}_{KYC}$, is valid at the claim epoch, and is not revoked, without revealing the concrete nationality value.

Fig. 2 illustrates how credential issuance and claim verification are connected. The KYC workflow starts with a setup phase. The issuer initializes an empty revocation tree (Step 1a), and the oracle committee publishes policy $\mathcal{P}_{KYC}$ on *BCTarget* (Step 1b). A user then registers with a trusted issuer and undergoes identity verification. Upon successful verification, the issuer generates a VC as defined in Definition 4.4 and returns it to the user (Step 2). During the burn transaction, the user computes VC_{hash} and includes it in the commitment hash.

When initiating a claim transaction on *BCTarget*, the user retrieves the current KYC policy $\mathcal{P}_{KYC}$ and the revocation-tree root (Step 3). The user then constructs a VP containing a ZKP of knowledge of a valid commitment hash and a PoNR (Step 4). This proves that the submitted

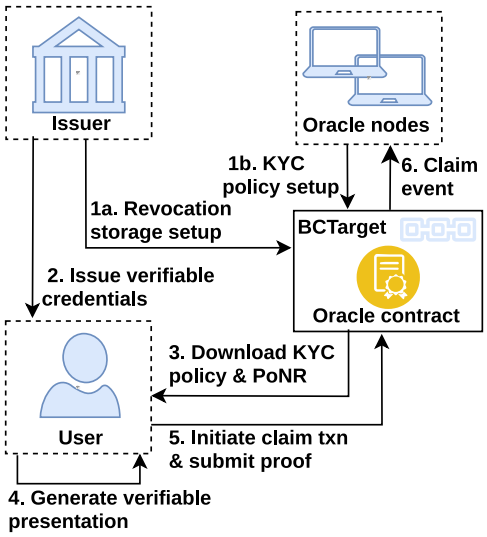


Fig. 2. KYC and credential verification workflow.

VC satisfies $\mathcal{P}_{KYC}$, is valid at the claim epoch, and is not revoked, without revealing the user's identity or concrete attribute values. The VP also includes VC_{hash} to bind the credential to the transaction and support revocation if misbehavior is detected. The user then submits the VP as part of the claim transaction (Step 5), after which the contract emits a claim event (Step 6).

After Step 6 in Fig. 2, enforcement continues through the validator-side logic in Fig. 1. Validators review each claim transaction by verifying the ZKP and checking for nullifier reuse (Fig. 1, Step 10). During aggregation, the aggregator finalizes the result once the validator votes satisfy a finalization threshold. If a request is invalid, its VC_{hash} is added to the oracle contract's revocation list, preventing further use of that credential. However, revoking the credential does not stop a malicious user from requesting a new one. To address this limitation, zkPACT adopts a hybrid SSI model that supports regulatory enforcement without compromising protocol-level privacy. With user consent, the issuer maintains an off-chain mapping between VC_{hash} and the user's real-world identity for legal or audit purposes only. If a credential is blacklisted for misbehavior, the corresponding VC_{hash} can be forwarded to the issuer, which denies reissuance to the same user. This preserves anonymity while supporting regulatory accountability in line with FATF's risk-based approach.

Implications for regulated financial BISs. zkPACT is well-suited to regulated financial BISs in which different organizations operate separate blockchain platforms, and users transfer tokenized values. A representative example is a transfer between two institutional platforms administered by different organizations but subject to the same compliance framework. In this setting, privacy matters as linking the source-side transfer to the target-side receipt can reveal payment behavior, business relationships, or asset positions to unnecessary parties. Compliance remains mandatory, as participation is limited to users who meet the eligibility requirements of the regulated environment.

zkPACT addresses this setting by allocating responsibilities across the transfer process. A trusted issuer, such as a licensed institution or a KYC provider, performs identity verification during onboarding and retains the audit record off-chain. During transfer, validators verify policy satisfaction and non-revocation without accessing the user's raw identity data, while the protocol preserves unlinkability between the source-side burn and target-side claim. As a result, organizations can support cross-platform transfers without each collecting the user's full identity record, while accountability is preserved through issuer-held audit records and credential revocation. This role allocation is

consistent with a system-based view of blockchain governance, which treats actors, responsibilities, incentives, and regulatory requirements as part of system design rather than as matters left entirely to external coordination [51].

4.3. Batching technique for efficient token claim verification

The following expands the claim verification and consensus phases in Fig. 1 (Steps 9–12) and shows how batching improves efficiency.

In zkPACT, each claim transaction requires off-chain proof verification by validators and on-chain confirmation by the oracle contract. Verifying claim requests individually would impose substantial off-chain computation and on-chain gas costs, especially as the number of requests grows. To address this limitation, zkPACT introduces a batching mechanism that groups multiple token claim requests into a batch, allowing a single aggregated proof to be generated for the entire batch.

To enable batching, the oracle contract on *BCTarget* assigns each claim transaction a unique deterministic ID and tracks the current round ID during claim-event submission (Fig. 1, Step 9). A round ID is a counter that identifies the current batch of token claim transactions. Validators follow a shared rule, encoded in the oracle contract, under which each round begins from the last finalized ID and includes the next b requests. Formally, for batch size b, round r maps to IDs in the range $[r \cdot b,\ (r+1) \cdot b - 1]$. This ensures that validators and the contract maintain a consistent view of batch membership even if some nodes observe events with slight delays.

Each validator encodes the batch decision as a single b-bit integer (a bitmask), where each bit represents a claim transaction. A bit is set to 1 if the validator accepts the corresponding proof and to 0 otherwise. This avoids handling separate votes for each request and reduces both computation and storage overhead. Validators then sign their bitmasks and submit them to the aggregator (Fig. 1, Step 10). The aggregator determines the batch outcome by applying a Byzantine Fault Tolerance (BFT) threshold-supported decision rule to each bit position.

Formally, for each position $j \in \{0, \ldots, b-1\}$ corresponding to a claim request in the batch, the threshold-supported decision bit is defined as $vote_{\tau}[j] = 1 \iff \sum_{i \in V} \mathbf{1}[vote_{val,i}[j] = 1] \geq \tau$, where V denotes the set of validators in the committee of size n, $\tau = f+1$ is the BFT finalization threshold, $f = \lfloor (n-1)/3 \rfloor$, and $\mathbf{1}[\cdot]$ denotes the indicator function. Thus, a request is accepted in the batch decision only if at least τ validators support it.

After collecting enough signed votes to satisfy the BFT finalization threshold, the aggregator applies this rule to every request position and derives a single threshold-supported bitmask for the batch. It then generates one aggregated ZKP attesting to the correctness of the threshold-supported batch decision and the corresponding update of validator states, and submits it to the oracle contract on *BCTarget* for verification (Fig. 1, Step 11). Instead of verifying a separate proof for each request, the oracle contract verifies only one aggregated proof per batch (Fig. 1, Step 12). This improves throughput and cost-efficiency by reducing redundant computation, lowering on-chain interactions and gas costs. Additionally, the on-chain validator state tree is updated only once per batch rather than per request, further reducing the cost of state transitions. Validators aligned with the threshold-supported decision are rewarded, while those who deviate are penalized under the stake-weighted reputation model described below.

4.4. Incentive and dynamic slashing mechanism

The following extends incentive and slashing mechanisms in Fig. 1 (Steps 11–12) by showing how validator rewards and penalties are integrated into the system.

Decentralized oracle networks rely on economic incentives to promote honest behavior and active participation [23]. Validators invest computational resources and stake capital to process cross-chain requests, and without proper incentives, they would have no motivation to participate. To address this, zkPACT rewards validators for correct behavior and penalizes them for incorrect or dishonest actions.

During off-chain aggregation, the aggregator assigns rewards and generates a ZKP proving the correct update of validator states (Fig. 1, Step 11). The oracle contract verifies this proof and stores the new validator-state root (Fig. 1, Step 12), avoiding repeated on-chain interactions and enabling efficient, verifiable reward distribution. The aggregator receives a higher reward than validators to account for proof generation and transaction fees associated with final on-chain submission. Validators may not be rewarded in every round, particularly if their submissions are delayed or excluded because aggregation is finalized as soon as enough signed votes satisfy the BFT threshold. To ensure long-term fairness, zkPACT rotates the aggregator role among committee members using round-robin selection, giving all validators regular opportunities to earn both validator-level and aggregator-level rewards.

zkPACT uses a Proof-of-Stake (PoS) validator model in which validators must lock their stake in the *BCTarget* oracle contract to join the committee. This stake serves as both an economic commitment and a Sybil-resistance mechanism. The system also maintains a reputation score reflecting each validator's voting accuracy over time and a severity count that increases with each incorrect vote. Together, these capture both reliability and repeated misbehavior, allowing slashing penalties to escalate for validators that continue to vote incorrectly.

In our framework, each validator i is represented by the state tuple $st_i = (id_i, pk_i, bal_i, rep_i, sev_i)$, where id_i is the validator identifier, pk_i is the public key, bal_i is the validator's current stake balance, rep_i is the reputation score reflecting past performance, and sev_i is the severity count tracking incorrect votes, with $sev_i = 0$ for a correct vote and $sev_i \geq 1$ for an incorrect vote, where higher values indicate repeated mistakes. New validators are initialized with a baseline reputation (e.g., $rep_i = 50$). The reputation update is performed as follows. Let rep_{old} denote the current reputation, Δ_c the fixed increment for a correct vote, and Δ_i the base penalty for an incorrect vote. For a correct vote, the reputation is updated as $rep_{\text{new}} = rep_{\text{old}} + \Delta_c$. To ensure fairness and prevent validators with high reputation scores from remaining in the system without ongoing evaluation, reputation values can be periodically adjusted or normalized to maintain a balanced system and keep validators accountable over time.

If the vote is incorrect, the updated reputation is computed as $rep_{\text{new}} = \max\left(rep_{\text{old}} - \Delta_i \cdot sev_{\text{old}},\ 0\right)$. This mechanism imposes a penalty proportional to the severity count for incorrect votes, while the max function ensures the reputation never falls below zero. Only validators who submit incorrect votes are penalized; those not included in the aggregated result do not incur any penalty. For a correct vote, the validator's stake balance is increased by a fixed reward, denoted as Δ_S. In this approach, the updated stake is computed as $bal_{\text{new}} = bal_{\text{old}} + \Delta_S$.

For an incorrect vote, a slashing penalty is applied. Let α be the base slashing factor (e.g., 0.05) and β be the escalation factor (e.g., 0.3). The slashing factor is calculated as $\text{Slash\_Fct} = \alpha \cdot (1 + \beta\, sev_{\text{old}}) \left(1 - \frac{rep_{\text{old}}}{100}\right)$, and the stake is then updated as $bal_{\text{new}} = bal_{\text{old}} \cdot (1 - \text{Slash\_Fct})$. Here, the term $\left(1 - \frac{rep_{\text{old}}}{100}\right)$ ensures that a higher reputation results in a smaller penalty, while the factor $\left(1 + \beta\, sev_{\text{old}}\right)$ scales the penalty with the severity count. The idea is to avoid heavily punishing validators who usually behave correctly. Validators with a high reputation have demonstrated their reliability, so the system applies slightly smaller penalties. At the same time, the penalty increases if a validator continues to vote incorrectly, making repeated mistakes more costly. This design encourages consistent accuracy and discourages careless participation.

Finally, we combine stake and reputation using weighted averaging to compute the validator score, balancing economic commitment and historical reliability. This score determines who can join or remain in the committee. Let w_R and w_S denote the weights assigned to

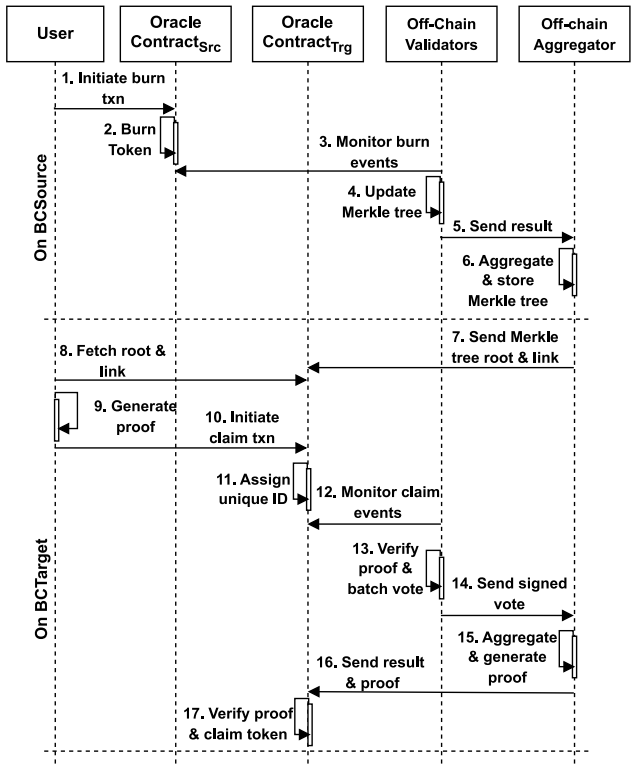


Fig. 3. Workflow of the system.

reputation and stake, respectively, such that $w_R + w_S = 1$. This normalization ensures the resulting validator score remains within a comparable range and reflects a weighted average of both components. We assign $w_R = 0.4$ and $w_S = 0.6$ to give slightly more influence to stake, acknowledging that financial commitment provides stronger Sybil resistance and economic security. The validator score is computed as ValidatorScore $= w_R\, rep_{\text{new}} + w_S\, bal_{\text{new}}$.

In our implementation, we use the following protocol parameter values to illustrate the incentive mechanism under a concrete setting. Each correct validator vote yields $\Delta_S = 2 \times 10^{10}$ and $\Delta_c = 2$, while the aggregator receives a higher reward of 5×10^{14} for completing the aggregation round. An incorrect validator vote incurs both reputation loss and stake slashing. For a validator with baseline reputation $rep = 50$, and using $\alpha = 0.05$ and $\beta = 0.3$, the slashing rate is 3.25% for the first incorrect vote, 4.00% for the second, and 4.75% for the third. For the first incorrect vote, the slashing loss exceeds the honest validator reward once the validator stake is above approximately 6.15×10^{11} stake units. Hence, honest voting is economically preferred whenever $bal_{\text{old}} \cdot \text{Slash\_Fct} > \Delta_S$. Under the parameter values used in zkPACT, this condition holds for sufficiently staked validators, so incorrect voting is more costly than honest participation from the first incorrect vote onward.

Validators can join or leave the system dynamically, enabling flexible participation while preserving decentralization. To join the committee, a new validator must exceed the validator score of the member it seeks to replace, preventing stake-only replacement and protecting reliable validators with strong histories. Exiting validators must complete a structured withdrawal process with a mandatory waiting period before reclaiming their stake, discouraging rapid exits and reentries and preserving committee stability. We next describe the system workflow during the token burn and claim phases.

4.5. System workflow

As shown in Fig. 3, the system workflow consists of two main phases: token burn on *BCSource* and token claim on *BCTarget*. In the figure, each entity is represented as a single component. The DFS and the issuer are not depicted, as the DFS primarily serves as off-chain storage, and it is assumed that users have already obtained their VCs (see Section 4.2). Additionally, validators are considered pre-registered as committee members in the oracle contract on *BCTarget*. All hashes are computed using an arithmetization-friendly hash function [52].

Token burn on the source blockchain. The process begins when a user invokes the burn function on *BCSource* and submits a commitment hash (see Definition 4.1) (Step 1). Upon receiving the burn transaction, the oracle contract verifies the balance and burns the specified token (Step 2). The contract then emits a burn event containing commitment hash c (Step 3). This event is monitored by validators across all participating blockchains. When validators receive burn events from multiple blockchains, they process them collectively. Although each blockchain maintains its own commitment tree, these trees include burn events from all participating blockchains. Thus, through cross-chain mixing, zkPACT increases the anonymity set, i.e., the number of commitment hashes among which a given one can be hidden.

Upon detecting a burn event, validators verify its inclusion in a valid block on *BCSource* and append the commitment hash c to their local commitment tree (Step 4). To maintain consistency across all nodes, validators wait for a fixed time window to collect concurrent burn events, including those from different blockchains, and then order them deterministically in lexicographic order by hash value. This ensures a uniform processing sequence and a consistent view of the commitment tree across oracle nodes. Following this, each validator submits the updated commitment-tree root and corresponding leaves to the current aggregator (Step 5). The aggregator verifies that enough validators support the same root to satisfy the BFT threshold and, once consensus is reached, stores the verified commitment-tree leaves in the DFS (Step 6). It then transmits the latest root and the DFS link to the oracle contract on *BCTarget* (Step 7). When a node joins the oracle committee or restarts after downtime, it rebuilds the local commitment tree from the latest DFS state to maintain consistency across validators.

Token claim on the target blockchain. To claim the burned tokens on *BCTarget*, the user first fetches the latest commitment-tree root and DFS link from the oracle contract (Step 8). The user verifies that the root matches the root recorded on the oracle contract, ensuring data integrity. After confirming the authenticity of the commitment tree, the user reconstructs it locally and generates a ZKP using the Claiming circuit (see Algorithm 1), demonstrating knowledge of the commitment, and a valid VC while keeping them confidential (Step 9). The user then submits a claim transaction to the oracle contract on *BCTarget* (Step 10). Additionally, the user pays a fee to incentivize oracle nodes to process the claim transaction.

Upon receiving the claim transaction, the oracle contract on *BCTarget* assigns a unique request ID (Step 11) and emits a claim event, notifying validators to initiate the verification process (Step 12). The validators verify the claim request by checking that nul_{hash} has not been previously recorded, ensuring no double-spending occurs. If these checks are satisfied, validators verify the ZKP in a batch (Step 13) and proceed with vote submission. Once the batch has been populated, all validators sign and submit their votes (Step 14). At this point, each validator queries the oracle contract to determine whether it has been assigned the aggregator role for the current round. The aggregator is selected using a round-robin mechanism to ensure fair distribution of responsibilities and prevent centralization.

The designated aggregator collects the votes, verifies validator signatures, applies rewards and slashing off-chain, and generates a ZKP. The proof is constructed using the Voting circuit (see Algorithm 2) for

Table 2
Circuit notation used in zkPACT.

Symbol	Description	Symbol	Description
nul	User nullifier	nul_{hash}	Hash of the nullifier
sec	User secret	$dest_{id}$	Destination-chain identifier
VC	Verifiable credential of the user	VC_{hash}	Hash of the credential
$leaf_{idx}$	Index of the commitment in the commitment tree	$commit_{root}$	Root of the commitment tree
$commit_{path}$	Merkle path in the commitment tree	rev_{path}	Merkle path in the revocation tree for non-revocation of VC
pk_{issuer}	Issuer public key	rev_{root}	Root of the revocation tree
$epoch_{claim}$	Claim epoch	req_{ids}	Claim request identifiers in the batch
$batch_{commit}$	Commitment to batch request identifiers	$round_{id}$	Current batching round identifier
$vote^{*}_{\tau}$	Threshold-supported vote bitmask	$vote_{val}$	Validator vote bitmask
$bits_{val}$	Validator participation bitmask	$bits_{hon}$	Validator honesty bitmask
agg_{id}	Aggregator identifier	val_{id}	Validator identifier
sk_{agg}	Aggregator secret key	sig_{val}	Validator signature
pk_{agg}	Aggregator public key	pk_{val}	Validator public key
agg_{path}	Aggregator Merkle path	val_{path}	Validator Merkle path
bal	Stake balance state	rep, sev	Reputation and severity state
thr_{cnt}	Number of validators supporting $vote_{\tau}$	$state^{new}_{root}$	Updated validator-state root

* $\tau = f + 1$ denotes the BFT threshold, where $f = \lfloor (n-1)/3 \rfloor$ for committee size n.

Algorithm 1 Claiming circuit

Private Input: nul, sec, VC, rev_{path}, $dest_{id}$, $leaf_{idx}$, $commit_{path}$
Public Input: $commit_{root}$, nul_{hash}, VC_{hash}, pk_{issuer}, rev_{root}, $epoch_{claim}$
Ensure: Correctness of the token claim request and KYC verification.

1: **function** VerifyClaim($commit_{root}$, nul_{hash}, VC_{hash}, pk_{issuer}, rev_{root}, $epoch_{claim}$)
2: **assert** VerifyMerkleProof($leaf_{idx}, commit_{path}, commit_{root}$)
3: $nul^{comp}_{hash} \leftarrow$ Hash(nul)
4: **assert** $nul_{hash} = nul^{comp}_{hash}$
5: $VC^{comp}_{hash} \leftarrow$ Hash(VC)
6: **assert** $VC_{hash} = VC^{comp}_{hash}$
7: **assert** VerifyPoNR($rev_{path}, VC_{hash}, rev_{root}$)
8: **assert** VerifySignature(VC, pk_{issuer})
9: **assert** GetExpiryEpoch(VC) $\geq epoch_{claim}$
10: $commit^{comp}_{hash} \leftarrow$ Hash($nul, sec, dest_{id}, VC_{hash}$)
11: **assert** $commit_{path}[0] = commit^{comp}_{hash}$

reward distribution and the Slashing circuit (see Algorithm 3) for penalties, thereby attesting to correct batch verification and validator-state updates (Step 15). The proof, aggregated voting results, and updated validator states are then submitted to the oracle contract on *BCTarget* (Step 16). Once verified, the oracle contract mints the requested tokens for the batch and updates the validator-state tree (Step 17). The following describes the circuits that ensure correctness, compliance, and validator accountability.

4.6. Proof of off-chain computational correctness

In zkPACT, three circuits ensure security, privacy, and accountability in cross-chain transactions. The Claiming circuit (Algorithm 1) verifies user eligibility and compliance, while the Voting (Algorithm 2) and Slashing (Algorithm 3) circuits ensure vote integrity by rewarding honest validators and penalizing misbehavior. The circuit notation used in the following algorithms is summarized in Table 2.

Claiming circuit. This circuit (Algorithm 1) is used by the user, as the claimant, to generate a proof of a valid token claim while proving compliance with KYC requirements (see Section 4.2). The generated proof is verified off-chain by the validators.

Inputs. The private inputs are nul (user nullifier), sec (user secret), VC (user VC), rev_{path} (proof of non-revocation for VC), $dest_{id}$ (destination-chain identifier), and a Merkle witness for the commitment tree consisting of the leaf index $leaf_{idx}$ and the Merkle path $commit_{path}$. The public inputs are $commit_{root}$ (root of the commitment tree), nul_{hash} (hash of the nullifier), VC_{hash} (hash of the credential), pk_{issuer} (issuer public key), rev_{root} (root of the revocation tree), and $epoch_{claim}$ (claim epoch).

Verification. The circuit first verifies inclusion of the user's commitment in the commitment tree by checking the leaf index $leaf_{idx}$ and the Merkle path $commit_{path}$ against the public root $commit_{root}$ (Line 2). This ensures that the claim is linked to a valid burn commitment recorded in the tree. It then computes the hash of the user nullifier nul and verifies equality with the public nullifier hash nul_{hash} (Lines 3–4), thereby binding the claim to a unique nullifier and preventing double-spending. The circuit then hashes the credential VC and verifies equality with the public credential hash VC_{hash} (Lines 5–6). This ensures consistency between the submitted credential and the credential bound to the claim. Next, the circuit verifies non-revocation of VC by checking the Merkle path rev_{path} against the public revocation root rev_{root} (Line 7). This ensures that the credential has not been revoked. The circuit further verifies the issuer's signature on VC using the public key pk_{issuer} (Line 8), thereby confirming that the credential was issued by a trusted issuer.

The circuit also verifies the expiration of the VC by checking that its expiry epoch is not earlier than the public $epoch_{claim}$ (Line 9). This ensures that the credential remains valid at the claim epoch. Finally, the circuit recomputes the commitment hash from nul, sec, $dest_{id}$, and VC_{hash} (Line 10) and verifies that it matches the first element of the Merkle path $commit_{path}$ (Line 11). This ensures that the claim is bound to the exact commitment originally created at burn time and to the intended destination chain. These checks guarantee that only users with valid credentials and commitments can claim tokens.

Voting circuit. Algorithm 2 is used by the aggregator to generate a proof that the published threshold-supported vote over a batch of claim requests and the corresponding validator-state update are correct. The circuit ensures that only valid votes signed by eligible validators are included in the aggregation, and that the resulting state transition is correctly computed. The generated proof is verified on-chain by the oracle contract.

Inputs. The private inputs are sk_{agg} (aggregator secret key), bal_{agg} (aggregator balance), rep_{agg} (aggregator reputation), sev_{agg} (aggregator severity count), agg_{path} (aggregator Merkle path), and req_{ids} (claim request identifiers). For each validator, the private inputs are pk_{val} (validator public key), sig_{val} (validator signature), bal_{val} (validator balance), rep_{val} (validator reputation), sev_{val} (validator severity count), $vote_{val}$ (validator vote bitmask), and val_{path} (validator Merkle path). The public inputs are $state^{new}_{root}$ (updated validator-state root), $batch_{commit}$ (commitment to batch identifiers), $vote_{\tau}$ (threshold-supported vote bitmask), $round_{id}$ (batch round identifier), $bits_{val}$ (validator participation bitmask), and $bits_{hon}$ (validator honesty bitmask).

Verification. The circuit first checks that participating validators have distinct identifiers (Line 2), ensuring that each validator is counted only once. It then recomputes the batch commitment from the private request identifiers req_{ids} and verifies equality with the public value

Algorithm 2 Voting circuit

Private Input: sk_{agg}, bal_{agg}, rep_{agg}, sev_{agg}, agg_{path}, req_{ids}, pk_{val}, sig_{val}, bal_{val}, rep_{val}, sev_{val}, $vote_{val}$, val_{path}
Public Input: $state^{new}_{root}$, $batch_{commit}$, $vote_{\tau}$, $round_{id}$, $bits_{val}$, $bits_{hon}$
Ensure: Correctness of validator votes and the state root update.

1: **function** VERIFY($state^{new}_{root}$, $batch_{commit}$, $round_{id}$, $vote_{\tau}$, $bits_{val}$, $bits_{hon}$)
2: **assert** UNIQUEVALIDATORIDS
3: $batch^{comp}_{commit} \leftarrow$ HASH(req_{ids})
4: **assert** $batch^{comp}_{commit} = batch_{commit}$
5: **assert** VERIFYMERKLEPROOF(agg_{path})
6: **assert** VERIFYKEY(sk_{agg}, agg_{path})
7: **Update Aggregator State:** bal_{agg}, rep_{agg}, sev_{agg}
8: **Update** $agg_{path}[0] \leftarrow$ HASH($agg_{id}, pk_{agg}, bal_{agg}, rep_{agg}, sev_{agg}$)
9: $state^{comp}_{root} \leftarrow$ COMPUTEROOT(agg_{path})
10: $bits_{sum} \leftarrow 0$; $hon_{sum} \leftarrow 0$; $thr_{cnt} \leftarrow 0$
11: **for each** validator **do**
12: **assert** VERIFYMERKLEPROOF($val_{path}, state^{comp}_{root}$)
13: $msg_{val} \leftarrow$ HASH($val_{id}, batch_{commit}, vote_{val}, round_{id}$)
14: **assert** VERIFYSIGNATURE($sig_{val}, msg_{val}, pk_{val}$)
15: $match \leftarrow [\, vote_{val} = vote_{\tau} \,]$; $thr_{cnt} \leftarrow thr_{cnt} + match$
16: **Update Validator State:** bal_{val}, rep_{val}, sev_{val}
17: **Update** $val_{path}[0] \leftarrow$ HASH($val_{id}, pk_{val}, bal_{val}, rep_{val}, sev_{val}$)
18: $state^{comp}_{root} \leftarrow$ COMPUTEROOT(val_{path})
19: $bits_{sum} \leftarrow bits_{sum} + 2^{val_{id}}$; $hon_{sum} \leftarrow hon_{sum} + match \cdot 2^{val_{id}}$
20: **end for**
21: **assert** $thr_{cnt} \geq f + 1$, **where** $f = \left\lfloor \frac{n-1}{3} \right\rfloor$
22: **assert** $bits_{val} = bits_{sum}$
23: **assert** $bits_{hon} = hon_{sum}$
24: **assert** $state^{new}_{root} = state^{comp}_{root}$

Algorithm 3 Slashing circuit

Private Input: pk_{agg}, bal_{agg}, rep_{agg}, sev_{agg}, agg_{path}, req_{ids}, pk_{val}, sig_{val}, bal_{val}, rep_{val}, sev_{val}, $vote_{val}$, val_{path}
Public Input: $state^{new}_{root}$, $batch_{commit}$, $vote_{\tau}$, $round_{id}$
Ensure: Correct detection of validator misbehavior, correct penalty application, and correct state-root update.

1: **function** VERIFYSLASHING($state^{new}_{root}$, $batch_{commit}$, $round_{id}$, $vote_{\tau}$)
2: $batch^{comp}_{commit} \leftarrow$ HASH(req_{ids})
3: **assert** $batch^{comp}_{commit} = batch_{commit}$
4: **assert** VERIFYMERKLEPROOF(val_{path})
5: $msg_{val} \leftarrow$ HASH($val_{id}, batch_{commit}, vote_{val}, round_{id}$)
6: **assert** VERIFYSIGNATURE($sig_{val}, msg_{val}, pk_{val}$)
7: **assert** $vote_{val} \neq vote_{\tau}$
8: **Update Validator State:** bal_{val}, rep_{val}, sev_{val}
9: **Update** $val_{path}[0] \leftarrow$ HASH($val_{id}, pk_{val}, bal_{val}, rep_{val}, sev_{val}$)
10: $state^{comp}_{root} \leftarrow$ COMPUTEROOT(val_{path})
11: **assert** VERIFYMERKLEPROOF($agg_{path}, state^{comp}_{root}$)
12: **Update Aggregator State:** bal_{agg}, rep_{agg}, sev_{agg}
13: **Update** $agg_{path}[0] \leftarrow$ HASH($agg_{id}, pk_{agg}, bal_{agg}, rep_{agg}, sev_{agg}$)
14: $state^{comp}_{root} \leftarrow$ COMPUTEROOT(agg_{path})
15: **assert** $state^{new}_{root} = state^{comp}_{root}$

$batch_{commit}$ (Lines 3–4). This binds the proof and validator votes to a specific batch of claim requests. The circuit next authenticates the aggregator's membership in the validator-state tree using the Merkle path agg_{path} (Line 5). It then checks that the secret key sk_{agg} corresponds to the aggregator record authenticated by that path (Line 6), thereby ensuring that the proof is generated by the designated aggregator. After that, the circuit updates the aggregator state variables bal_{agg}, rep_{agg}, and sev_{agg}, updates the corresponding leaf in agg_{path}, and recomputes the resulting state root $state^{comp}_{root}$ (Lines 7–9). For details on how these state variables are updated, see Section 4.4. The circuit then initializes three accumulators: $bits_{sum}$ for the reconstructed participation bitmask, hon_{sum} for the reconstructed honesty bitmask, and thr_{cnt} for counting how many validators support the published threshold-supported vote bitmask (Line 10).

For each validator (Lines 11–20), the circuit verifies membership in the validator-state tree by checking the Merkle path val_{path} against the current computed root $state^{comp}_{root}$ (Line 12). This ensures that only registered validators are included. It then computes the signed vote message msg_{val} from the validator identifier, batch commitment, vote bitmask, and round ID, and verifies the signature using pk_{val} (Lines 13–14). This binds the vote to a specific validator, batch, and round, thereby preventing replay or vote reuse across different rounds or batches. The circuit next computes the honesty indicator $match = [\, vote_{val} = vote_{\tau} \,]$ and increments the counter thr_{cnt} accordingly (Line 15). This determines whether the validator's vote agrees with the published threshold-supported vote. Validators with $match = 1$ are treated as honest in the subsequent state update and are rewarded accordingly (Line 16) (see Section 4.4).

The circuit updates the validator state variables bal_{val}, rep_{val}, and sev_{val}, rehashes the corresponding leaf in val_{path} (Line 17), and recomputes the resulting state root $state^{comp}_{root}$ (Line 18). This root is carried forward across iterations to accumulate validator-state updates. It also reconstructs the public bitmasks by updating $bits_{sum}$ for participation and hon_{sum} for honesty through the terms $2^{val_{id}}$ and $match \cdot 2^{val_{id}}$, respectively (Line 19). This records which validators participated in the round and which of them voted consistently with $vote_{\tau}$. After processing all validators, the circuit checks that the number of validators supporting $vote_{\tau}$ satisfies $thr_{cnt} \geq \left\lfloor \frac{n-1}{3} \right\rfloor + 1$ (Line 21), where n is the committee size. This confirms sufficient support for the published threshold-supported vote. It then verifies $bits_{val} = bits_{sum}$ and $bits_{hon} = hon_{sum}$ (Lines 22–23), ensuring that the aggregator cannot omit a participating validator, add a non-participating one, or misreport which validators agreed with $vote_{\tau}$. Finally, it verifies that the claimed updated root $state^{new}_{root}$ equals the computed root $state^{comp}_{root}$ (Line 24), so that all validator-state updates are reflected in the published final state root.

Slashing circuit. Algorithm 3 is used by the aggregator to generate a proof that a validator's signed vote differs from the published threshold-supported vote $vote_{\tau}$ and that the corresponding penalty and validator-state update were applied correctly. The generated proof is verified on-chain by the oracle contract.

Inputs. The Slashing circuit uses the same core private inputs as the Voting circuit, namely pk_{agg}, bal_{agg}, rep_{agg}, sev_{agg}, agg_{path}, and req_{ids}, together with, for the penalized validator, pk_{val}, sig_{val}, bal_{val}, rep_{val}, sev_{val}, $vote_{val}$, and val_{path}. The public inputs are $state^{new}_{root}$, $batch_{commit}$, $vote_{\tau}$, and $round_{id}$.

Verification. The circuit first recomputes the batch commitment from the private request identifiers req_{ids} and verifies equality with the public value $batch_{commit}$ (Lines 2–3). This binds the slashing proof to a specific batch and round. It then verifies membership of the validator in the validator-state tree by checking the Merkle path val_{path} (Line 4). Next, it reconstructs the signed vote message msg_{val} from the validator identifier, batch commitment, vote, and round identifier, and verifies the signature using pk_{val} (Lines 5–6) to confirm the vote was issued by that validator for the given batch and round.

The circuit then checks that the validator's vote differs from the published threshold-supported vote $vote_{\tau}$ (Line 7). This identifies the validator as misbehaving. It next updates the validator state variables bal_{val}, rep_{val}, and sev_{val} to apply the corresponding penalty (Line 8). Details of these update rules are given in Section 4.4. The circuit then rehashes the validator leaf in val_{path} and recomputes the resulting state root $state^{comp}_{root}$ (Lines 9–10). This records the validator penalty in the validator-state tree. Next, it verifies the aggregator's membership by checking the Merkle path agg_{path} against the updated root (Line 11), and then updates the aggregator state variables bal_{agg}, rep_{agg}, and sev_{agg} (Line 12). It rehashes the aggregator leaf and recomputes the final root (Lines 13–14). Finally, it verifies that the computed root equals the claimed updated root $state^{new}_{root}$ (Line 15), ensuring the integrity of updates.

5. Implementation

We provide open-source proof-of-concept implementations for both zkPACT and the zkOracle baseline on GitHub. [1] Our implementation is developed in Go version 1.24.0 and uses go-ethereum[2] to interact with the blockchain via JSON-RPC, ABI encoding, and transaction signing, with bindings generated using `abigen`. The oracle backend exposes a REST interface that maps client requests to protocol operations and submits signed transactions to the connected chains. The smart contracts are implemented in Solidity (v0.8.30) and are built and deployed using Foundry.[3] For development, we run a local Ethereum execution client based on Reth (v0.2.0-beta.2) and interact with it through standard JSON-RPC. The system is designed to be blockchain-agnostic, supporting deployment across blockchains with native support for smart contracts and elliptic curve pairing operations. However, for our evaluation, we target Ethereum due to its mature tooling, widespread adoption, and support for cryptographic primitives essential for zk-SNARKs verification. For hashing, we use Minimum Multiplicative Complexity (MiMC) [52], a hash function designed for efficient use in arithmetic circuits.

5.1. Off-chain components

Each oracle node runs a Go-based backend service that manages communication, event processing, and the local state required for cross-chain interactions, and exposes a REST interface that maps client requests to protocol operations and submits signed transactions via JSON-RPC. The backend also includes modules for event indexing, batch formation, proof generation, and transaction submission, enabling protocol operations to be executed deterministically from chain events and committee inputs. A core component of the backend service is an append-only, fixed-depth incremental commitment tree used to record commitment hashes derived from burn transactions. Since new commitments are added frequently, this structure is optimized for handling high update rates. Each leaf can be added in $O(\log n)$ time, and Merkle proofs are generated with the same complexity. Initially, all leaves in the tree are set to zero and are sequentially replaced with non-zero commitments.

Each oracle maintains a local copy of the commitment tree and stays synchronized with others. The backend parallelizes event handling, batch construction, and verification routines using Go's concurrency primitives, such as goroutines and channels, to ensure efficient parallel processing. To support proof generation, the latest state of the commitment tree leaves is periodically published to the DFS, for efficient retrieval by users and validators without incurring additional on-chain storage or gas costs. A separate index is also maintained on the DFS, allowing users to locate their commitment in $O(1)$ time and generate Merkle inclusion proofs efficiently. In our implementation, InterPlanetary File System (IPFS) serves as the DFS.

To support KYC functionality, our implementation includes an incremental, fixed-depth Merkle tree maintained off-chain by the issuer as a revocation tree. Each leaf corresponds to a user credential indexed at a fixed position. In zkPACT, the credential encodes a single KYC attribute, and users generate a ZKP of non-revocation and policy compliance using the Claiming circuit (see Algorithm 1).

5.2. On-chain components

The core on-chain logic is encapsulated in the oracle contract, coordinating the selection of the next aggregator, the aggregator's vote submission, validator registration, stake withdrawal, replacement, and exiting. The oracle contract integrates a separate MerkleTree contract that implements a sparse Merkle tree for validator state. This tree tracks validator accounts, including balances and reputation-related metadata, and remains on-chain because it governs security-critical operations such as membership updates, stake withdrawals, rewards, replacements, and slashing, which must be enforced directly by the oracle contract without additional trust or data-availability assumptions. Updates to this state require on-chain Merkle proof verification, ensuring the integrity of validator actions. A sparse Merkle tree is chosen because updates occur only once per batch and each validator has a fixed position in the tree, thereby avoiding the need for structural modifications during updates.

The oracle contract is deployed alongside four supporting contracts: the MerkleTree contract, the votingVerifier, the slashingVerifier, and a dedicated MiMC hashing contract. The verifier contracts are automatically generated using the gnark library[4] to support efficient ZKP verification on Ethereum. The votingVerifier contract verifies ZKPs generated by the Voting circuit (see Algorithm 2), ensuring that validator votes have been correctly processed and state updates follow protocol rules. The slashingVerifier contract checks ZKPs from the Slashing circuit (see Algorithm 3), which enforces dynamic, reputation-based penalties for dishonest validators.

For comparison, we reimplement zkOracle [23] as the evaluation baseline because it is the closest oracle-based cross-chain solution to zkPACT and adopts a non-batched verification model. To ensure a fair comparison, we implement the baseline in the same environment as zkPACT, including the Go backend, smart-contract toolchain, and local execution setup, while preserving zkOracle's original verification flow, non-batched processing model, and circuits. This setup ensures that the measured differences reflect the protocol design and processing model rather than deployment choices.

6. Evaluation

To assess the effectiveness of zkPACT, we compare it with zkOracle [23] in terms of memory and gas consumption, position its per-transfer gas cost against related ZKP-based cross-chain systems, and finally report throughput and latency under concurrent workloads.

6.1. Experimental setup

Experiments were conducted on a virtual machine running Ubuntu 22.04.4 LTS, equipped with an AMD EPYC Milan processor featuring 16 vCPUs, 64 GB of RAM, and 360 GB of SSD storage. We use Foundry (forge v1.3.5) to compile, deploy, and test the contracts in a local Ethereum environment. We evaluate committee sizes $n \in \{4, 8, 16, 32, 64, 128\}$ and batch sizes $b \in \{1, 5, 10, 15, 20\}$. The committee sizes were chosen for two protocol-driven reasons. First, zkPACT maintains validator states in a Merkle tree, so using committee sizes that grow as powers of two yields balanced tree structures and provides a clean way to study how tree-based state updates scale as the committee expands. Second, the Voting circuit encodes validator participation and voting outcomes as BN254 bitmasks. Since one BN254 field element can represent up to 254 validator positions, our largest setting remains safely within this bound. Moreover, $n = 128$ already exceeds committee sizes typically reported for real-world decentralized oracle settings [53]. Supporting $n > 254$ would only require extending

[1] https://github.com/ElmiraEbrahimi/zkPACT.git
[2] https://github.com/ethereum/go-ethereum (v1.16.7)
[3] https://getfoundry.sh/
[4] https://github.com/consensys/gnark (v0.14.0)

the encoding across multiple field elements, rather than changing the circuit design itself.

For zkPACT, the case $b = 1$ represents non-batched execution and therefore provides the direct reference point for comparison with zkOracle. The settings $b \in \{5, 10, 15, 20\}$ then show how batching changes off-chain and on-chain costs relative to that non-batched approach. The reported range was selected to capture the transition from non-batched execution to effective batched operation without adding redundant configurations. Each configuration was run 50 times, and the mean and standard deviation are reported to capture performance consistency. Across configurations, the standard deviations remained low, ranging from 0.01 GB to 0.25 GB for memory usage and within 2%–5% of the mean for gas consumption. These variances indicate consistent performance and support the reliability of the observed trends.

6.2. Off-chain computational cost

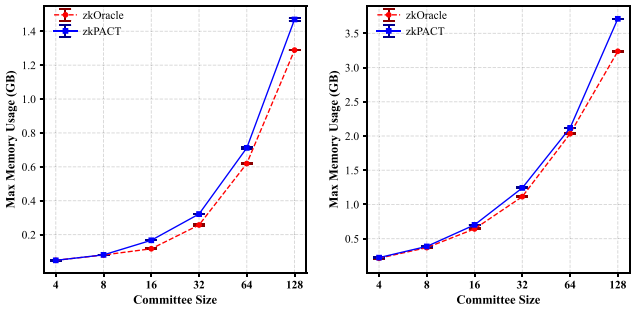
This section evaluates off-chain computational burdens by measuring peak memory consumption during circuit compilation and proof generation for the three core circuits presented in Section 4.6. We compare these results with zkOracle to highlight the performance improvements.

Users leverage the Claiming circuit to generate a ZKP proving knowledge of a valid commitment hash and possession of a non-revoked credential satisfying the verifier's KYC policy. Since the circuit handles individual claim requests, its structure is independent of batch and committee size. Accordingly, its off-chain cost remains constant across configurations. Specifically, Claiming-circuit compilation, proof generation, and off-chain verification require approximately 50 MB, 66 MB, and 39 MB, respectively. These stable requirements keep user-side interactions lightweight and predictable.

We conducted three experimental setups to evaluate the off-chain computational cost of the Voting circuit. In each setup, we varied the batch or committee sizes while keeping the other parameter constant to isolate the impact on memory usage during circuit compilation and proof generation.

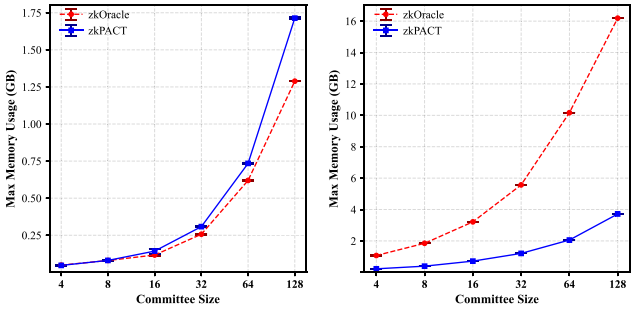
Voting circuit peak memory at fixed batch size ($b = 1$). The results of this configuration are shown in Fig. 4. This setup reflects a non-batched system that generates separate proofs for each request. As the number of validators increases, memory usage rises in both systems' compilation and proof generation phases due to the growing number of signature checks and Merkle path verifications. In this setup, the zkPACT Voting circuit uses more memory than zkOracle in both the compilation phase (Fig. 4(a)) and the proof generation phase (Fig. 4(b)). At a committee size of 128, the additional overhead remains moderate, with compilation memory increasing from approximately 1.29 GB in zkOracle to 1.47 GB in zkPACT, and proof-generation memory increasing from approximately 3.24 GB to 3.71 GB. This moderate increase is due to additional constraints required to support incentive enforcement and accountability, which make the zkPACT Voting circuit more complex than the zkOracle circuit. While memory usage scales linearly with circuit-dependent parameters, the superlinear growth observed beyond 64 validators is due to our experimental setup. To simulate a real-world deployment, we run components in parallel within a shared-memory environment. This shared context introduces additional overhead that does not occur in real-world settings, where independent execution would preserve linear scalability of zk-SNARK-based designs.

Voting circuit peak memory at fixed batch size ($b = 5$). The results of this experiment are shown in Fig. 5, which evaluates the efficiency of batching in zkPACT when handling multiple requests. In this setup, zkPACT exhibits higher memory usage than zkOracle during compilation (Fig. 5(a)) due to additional constraints introduced by batch processing. This compilation cost represents a one-time overhead for validators and has a limited long-term impact. Among the factors affecting system scalability, memory usage during proof generation is the most critical,



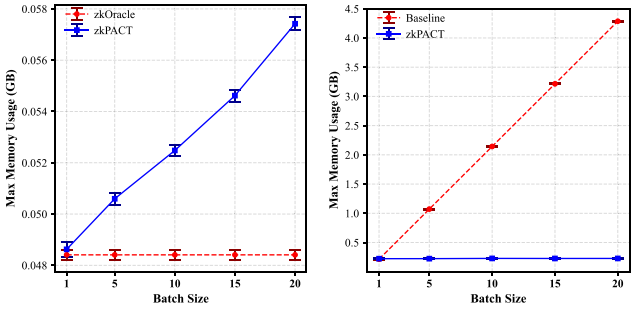
(a) Compile memory (b) Proof generation memory

Fig. 4. Voting circuit peak memory: zkPACT vs. zkOracle at fixed batch size $b = 1$ across varying committee sizes.



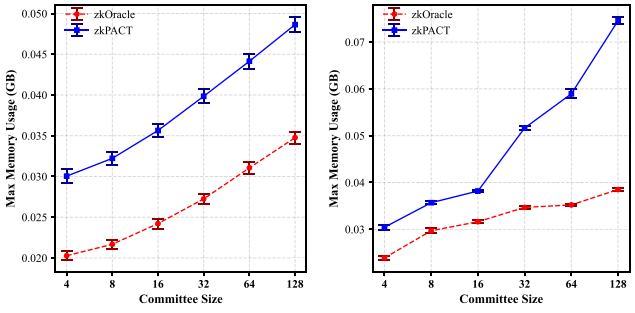
(a) Compile memory (b) Proof generation memory

Fig. 5. Voting circuit peak memory: zkPACT vs. zkOracle at fixed batch size $b = 5$ across varying committee sizes.



(a) Compile memory (b) Proof generation memory

Fig. 6. Voting circuit peak memory: zkPACT vs. zkOracle at fixed committee size $n = 4$ across varying batch sizes.



(a) Compile memory (b) Proof generation memory

Fig. 7. Slashing circuit peak memory: zkPACT vs. zkOracle at fixed batch size $b = 1$ across varying committee sizes.

and zkPACT demonstrates a clear advantage, as shown in Fig. 5(b). While zkOracle generates a separate proof for each request, increasing memory usage with the committee size, zkPACT produces a single, aggregated proof for the entire batch. This results in significantly lower memory usage. For example, at a committee size of 128, zkPACT reduces proof-generation memory usage from approximately 16.19 GB in zkOracle to 3.72 GB, corresponding to about 4.3 times lower memory usage and a 77% reduction. These findings highlight the efficiency of batching in reducing off-chain computational costs during batched claim processing.

Voting circuit peak memory at fixed committee size ($n = 4$). The results of this configuration are shown in Fig. 6, illustrating the batching efficiency under increasing workloads. The zkOracle system exhibits constant memory usage during compilation (Fig. 6(a)) as its circuit is independent of the batch size. In contrast, zkPACT's compilation memory increases with batch size due to the growing number of constraints introduced by the batching technique. Regarding proof generation, the difference becomes substantial (Fig. 6(b)). Although both systems exhibit increasing memory usage with batch size, zkOracle grows significantly faster, even with a small committee size. In contrast, zkPACT produces a single proof for the entire batch, resulting in significantly lower memory usage. At a batch size of 20, zkPACT reduces proof-generation memory usage from approximately 4.29 GB in zkOracle to 0.23 GB, corresponding to about 18.9 times lower memory usage and a 94% reduction. As the committee size increases, this gap grows further, highlighting how batching improves scalability and reduces per-request overhead.

Slashing circuit peak memory at fixed batch size ($b = 1$). The results of this configuration are shown in Fig. 7. The Slashing circuit consistently uses less memory than the Voting circuit, as it involves fewer constraints and a simpler structure. Similar to the Voting circuit, zkPACT exhibits higher memory usage than zkOracle during both the compilation and proof generation phases (Fig. 7(a) and Fig. 7(b)). This increase is due to the additional constraints required to support dynamic slashing, introducing more complex logic than zkOracle. At a committee size of 128, this overhead remains moderate, with compilation memory increasing from approximately 0.035 GB in zkOracle to 0.049 GB in zkPACT, and proof-generation memory increasing from approximately 0.038 GB to 0.075 GB. Although zkPACT supports integrating Slashing and Voting circuits into a single circuit, we evaluate them separately to enable a fair comparison with zkOracle, where the slashing mechanism is implemented as an independent component, and the reported memory usage reflects the cost per request.

6.3. On-chain computational cost

Efficient use of on-chain resources is crucial for the practicality of DApps, as users must pay transaction fees based on the amount of gas consumed. In Ethereum, gas measures the computational and storage effort required to execute transactions, directly impacting user cost. zkPACT consists of several smart contracts. The oracle contract, which handles core protocol logic and validator coordination, requires 4,145,326 gas to deploy. The votingVerifier and slashingVerifier contracts, which act as on-chain verifiers for aggregating votes and slashing proofs, consume 2,155,006 and 2,022,676 gas, respectively, upon deployment. The MiMC contract adds 411,410 gas. These deployment costs are incurred once during setup and do not affect runtime operation.

Per-call gas at fixed batch size ($b = 1$). The measured gas usage of each core contract function under increasing committee sizes is shown in Fig. 8. Functions such as `burn` and `claim` exhibit nearly constant gas usage, averaging approximately 30,486 and 427,396 gas, respectively. The low cost of `burn` reflects its simplicity, as it records a commitment hash and emits an event. In contrast, `claim` is more expensive as it involves submitting a ZKP tied to a prior burn action. Aggregator-side functions, such as `submitWiVote` and `slash`, which submit proofs generated by the Voting and Slashing circuits, also maintain nearly constant gas consumption, averaging approximately 428,455 and 299,995 gas, respectively. This consistency arises from using fixed-size public inputs that are independent of the committee size.

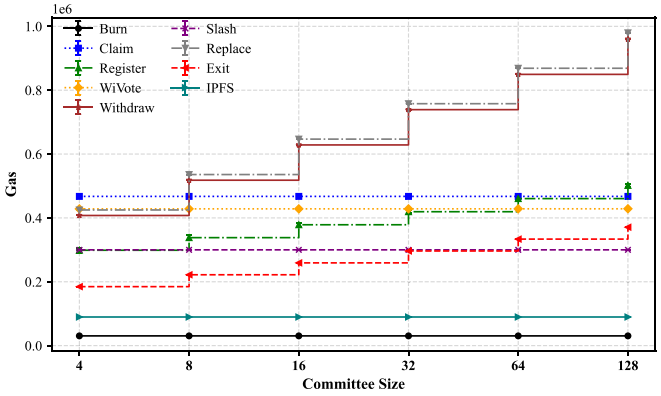


Fig. 8. Gas consumption per contract function call at fixed batch size $b = 1$ across varying committee sizes.

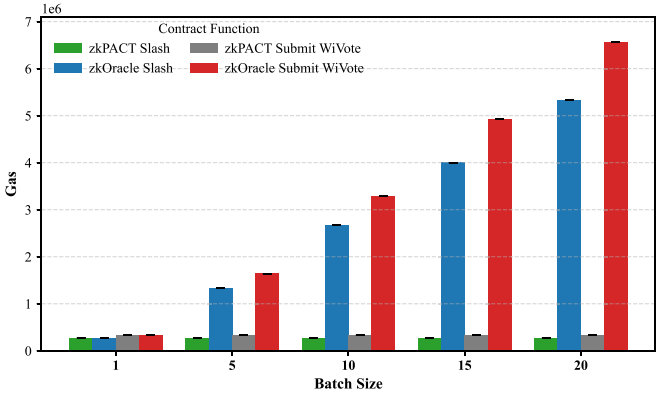


Fig. 9. Gas consumption: zkPACT vs. zkOracle at fixed committee size $n = 4$ across varying batch sizes.

Validator-side functions, such as `register`, `replace`, `exit`, and `withdraw`, exhibit increasing gas usage as the committee size increases. These functions perform Merkle proof verification, so their gas cost depends directly on the depth of the validator-state tree, which increases with the number of validators. For example, `register` consumes approximately 298,721 gas at a committee size of four and increases to 501,663 gas at a committee size of 128. At a committee size of 128, `withdraw` and `replace` consume approximately 960,080 and 979,681 gas, respectively, due to account updates, fund transfers, and proof verification. The `exit` function, which does not require fund transfers, consumes approximately 184,713 gas and grows to 370,838 gas. Although these operations are expensive, they are one-time costs for validators. Moreover, real-world oracle committees range from a few to several dozen nodes [53], and the reported values reflect conservative upper bounds.

Gas comparison at fixed committee size ($n = 4$). As shown in Fig. 9, we compare the gas consumption of the `submitWiVote` and `slash` functions to assess scalability under increasing load. Both functions involve on-chain verification of ZKPs generated by the Voting and Slashing circuits. The gas cost depends on the verification's computational complexity and the number of public inputs submitted to the verifier. In zkPACT, each batch of requests is verified using a single aggregated proof, and the number of public inputs sent to the on-chain verifier remains fixed regardless of batch size. As a result, the

verification cost remains constant across all batch sizes. In contrast, zkOracle performs one on-chain verification per request, resulting in a total gas consumption that increases proportionately to the number of requests. Overall, zkPACT achieves up to 95% lower gas usage than zkOracle, demonstrating its efficiency for high-throughput cross-chain interactions.

End-to-end per-transfer gas comparison. To show how batching reduces the cost of a complete cross-chain transfer, we first compare the per-transfer gas cost of zkPACT with our reimplemented zkOracle baseline, and then with other ZKP-based cross-chain systems. For zkPACT at $n = 4$ and $b = 20$, the measured end-to-end gas cost per transfer is $30,486 + 451,135 + 428,379/20 \approx 503,040$ gas, where 30,486 is the source-chain `burn` cost, 451,135 is the target-chain `claim` cost, and $428,379/20$ is the amortized per-transfer share of the batch-finalization transaction `submitWiVote`. The division by 20 reflects that one `submitWiVote` transaction finalizes a batch of 20 claim requests. For zkOracle at $n = 4$, the corresponding per-transfer cost from our reimplementation is $30,487 + 467,395 + 428,354 = 926,236$ gas, since verification is non-batched and the finalization cost is incurred separately for each transfer. This corresponds to an end-to-end gas reduction of approximately 45.7% for zkPACT relative to zkOracle.

We also compare zkPACT with existing ZKP-based cross-chain systems using their reported per-transfer gas values. Compared with Hu et al. [48], whose aggregated ZKP-based cross-chain scheme reports 848,969 gas per transaction, zkPACT at $b = 20$ is 40.7% lower. Compared with zkCross [11], zkPACT is approximately 44.2% lower than its reported cross-chain exchange cost (901,472 gas) and approximately 1.8% higher than the reported gas cost of its cross-chain transfer protocol (494,000 gas). zkBridge [12] reports lower costs of roughly 210K–221K gas, depending on the operation reported; however, these costs correspond to trustless block-header relay and proof verification rather than to a privacy-preserving claim pipeline comparable to zkPACT. Likewise, Sober et al. [47] report much lower amortized burn and claim costs, namely 4606 and 4848 gas at a batch size of 128, but these values explicitly exclude cross-blockchain communication and are measured for a rollup-style transfer design without privacy, compliance, and accountability. Compared with Harmonia [27], zkPACT is approximately 79.7% higher than its reported single light-client update cost of about 280K gas, but approximately 47.0% lower than its reported state-migration cost of about 950K gas. This difference is consistent with the fact that Harmonia optimizes trust-minimized light-client update and migration operations, whereas zkPACT targets batched, privacy-preserving claim finalization with compliance and oracle accountability.

Overall, zkPACT reduces gas relative to zkOracle and remains competitive with other ZKP-based cross-chain systems while supporting private claims, compliance checks, and validator accountability.

6.4. Throughput and latency evaluation

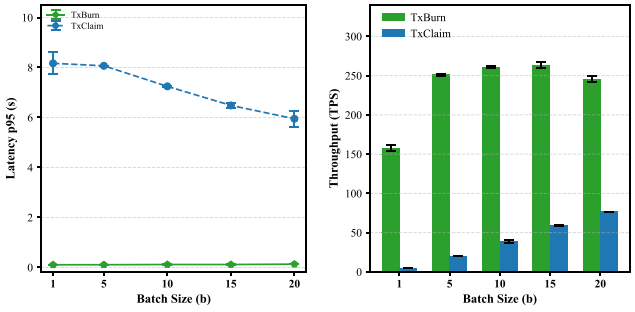
We evaluate zkPACT under concurrent workloads to highlight its processing capacity and the effects of batching and committee coordination on the burn and claim paths.

Benchmark design and workload generation. We evaluate throughput and latency for the burn and claim paths under controlled workloads. In each run, $p \in \{1, 4, 8, 16, 32\}$ parallel users submit requests concurrently, and each user issues b requests. The offered load is therefore $p \cdot b$ requests per run. For example, with $p = 32$ and $b = 20$, the system receives 640 concurrent requests, i.e., 640 burn requests in the burn workload or 640 claim requests in the claim workload.

We use $p = 32$ as the main reporting point as it is the highest stable concurrency supported by our experimental environment and therefore exposes bottlenecks most clearly. Lower-p runs were used to validate the measurement setup before applying the highest load. Throughput is measured as the number of successfully completed transactions per second on the evaluated path, and latency is measured as the end-to-end completion time on that path. We use p95 as the primary tail-latency metric under load and include p50 where relevant. Each (p, n, b) configuration was executed in five independent runs, with throughput averaged across runs and latency p50 and p95 computed from the aggregated per-request samples. For all configurations, the variability across runs remained low, supporting the consistency of the reported throughput and latency trends.

Table 3
Throughput and latency at fixed committee size $n = 4$.

b	TxBurn p95 (ms)	TxClaim p95 (ms)	TxBurn TPS	TxClaim TPS
1	105.51	8167.12	157.56	4.57
5	106.89	8067.49	251.34	19.67
10	115.61	7244.58	260.68	38.88
15	115.32	6482.41	263.26	58.75
20	130.44	5947.18	245.66	76.68



(a) Latency (b) Throughput

Fig. 10. Latency and throughput: zkPACT at fixed committee size $n = 4$ across varying batch sizes.

For the burn path, latency is measured from transaction submission to burn confirmation on *BCSource*. For the claim path, latency is measured from claim submission to on-chain finalization on *BCTarget*. The reported claim latency therefore captures the full processing path, including validator-side verification of the submitted proof, batch formation, aggregator-side generation of the off-chain computation proof, on-chain verification of the resulting batch proof, and final inclusion in a mined block. The user-side proof generation using the Claiming circuit is excluded from this latency because it is executed locally before claim submission, so we report it separately. Since the Claiming circuit is independent of n and b, its runtime remains effectively constant across configurations. In our measurements, compilation takes about 87 ms and proof generation takes about 101 ms. By contrast, validator-side verification of this submitted proof is part of the claim-processing path and is therefore included in the reported claim latency.

Throughput and latency at fixed committee size ($n = 4$). To examine how batching affects system behavior under load, we fix the committee size at $n = 4$ and vary the batch size over $b \in \{1, 5, 10, 15, 20\}$ in the highest-load configuration. Table 3 reports the corresponding throughput and p95 latency values.

For burn transactions (*TxBurn*), throughput increases from 157.56 TPS at $b = 1$ to 263.26 TPS at $b = 15$, then decreases to 245.66 TPS at $b = 20$ (Fig. 10(b)). In contrast, p95 latency changes only slightly, from 105.51 ms to 130.44 ms (Fig. 10(a)). Together, these results indicate an initial utilization gain followed by saturation. Larger b first improves source-chain pipeline occupancy, but beyond the peak region, further load does not translate into higher throughput. The modest latency change confirms that *TxBurn* remains a short-path operation with stable service time over this range.

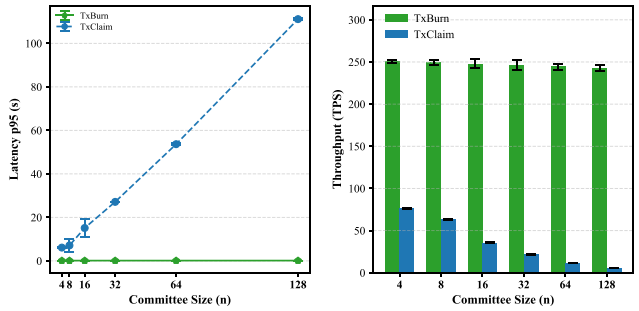
For claim transactions (*TxClaim*), throughput increases from 4.57 TPS at $b = 1$ to 76.68 TPS at $b = 20$ (Fig. 10(b)), while p95 latency decreases from 8167.12 ms to 5947.18 ms (Fig. 10(a)). At $b = 20$ and $p =$

Table 4
Claim throughput and latency at fixed batch size $b = 20$.

n	TxClaim p50 (ms)	TxClaim p95 (ms)	TxClaim TPS
4	4047.75	6110.84	76.50
8	4615.59	7113.27	63.25
16	8866.38	15 126.25	35.51
32	16 291.72	27 132.21	21.79
64	30 274.53	53 653.42	11.02
128	60 356.14	111 145.64	5.71

Table 5
End-to-end transfer latency at fixed committee size $n = 4$.

b	E2E p50 (ms)	E2E p95 (ms)
1	4637.62	7596.94
5	4491.65	8269.76
10	4251.93	7447.51
15	4253.02	6685.64
20	4161.03	6151.27



(a) Latency (b) Throughput

Fig. 11. Latency and throughput: zkPACT at fixed batch size $b = 20$ across varying committee sizes.

32, this throughput corresponds to approximately 3.83 finalized batches per second. This behavior is consistent with batched finalization under sustained concurrent load. In this setting, claim requests arrive quickly enough for batches to be filled continuously, while the fixed per-round costs of validator coordination, aggregator processing, and on-chain finalization are amortized across more claims as b increases, improving effective *TxClaim* throughput. The reported *TxClaim* results, therefore, also reflect the system's practical batch-finalization behavior. Under sustained load, this also reduces queueing pressure per claim in the measured region and lowers tail latency. Preliminary runs beyond the reported range confirm the same batching behavior. For example, at $n = 4$, increasing b to 25 raises claim finalization throughput to 81.927 TPS, indicating that the selected range already captures the relevant throughput trend.

Overall, these results show that batch size is a key performance parameter in zkPACT. Moderate-to-large b substantially improves claim-side efficiency, whereas burn-side throughput reaches a clear peak under high load.

Throughput and latency at fixed batch size ($b = 20$). To examine how committee size affects system behavior under load, we fix the batch size at $b = 20$ and vary the committee size over $n \in \{4, 8, 16, 32, 64, 128\}$ under the highest-load configuration. Table 4 reports the claim throughput and latency values.

For *TxBurn*, throughput remains nearly stable as the committee size increases, decreasing only from 250.69 TPS at $n = 4$ to 242.73 TPS at $n = 128$ (Fig. 11(b)). Likewise, p95 latency changes only slightly, from 127.38 ms to 139.74 ms (Fig. 11(a)). This behavior is expected because burn execution takes place on *BCSource* and is not directly governed by committee coordination. The small variation with n is therefore better explained by end-to-end execution effects, such as shared host resources, RPC contention, and scheduling interference, than by a direct dependence on committee size.

For *TxClaim*, p95 latency rises from 6110.84 ms at $n = 4$ to 111145.64 ms at $n = 128$ (Fig. 11(a)). Over the same range, throughput decreases from 76.50 TPS to 5.71 TPS (Fig. 11(b)). This experiment also captures committee-side coordination performance in practice, because increasing n changes the amount of validator work required before a claim batch can be finalized. Along the claim path, validators verify submitted proofs and vote on claim validity, while the aggregator collects and processes a larger set of validator responses, generates the aggregated proof, and submits the batch-finalization transaction. As a result, larger committees increase coordination overhead, aggregator-side computation, and the time required to complete on-chain batch finalization. The observed increase in *TxClaim* latency and decrease in throughput therefore quantify the operational overhead introduced by larger oracle committees.

Moreover, we report p50, which increases from 4047.75 ms to 60356.14 ms, distinguishing typical claim completion time from tail latency. The wider gap between p50 and p95 at larger n indicates that many claims are completed earlier, even as queueing delays become more pronounced. The reported claim latency is measured from the submission of the first claim assigned to a batch to its on-chain finalization, corresponding to the worst-case user waiting scenario. The first claim in a batch waits the longest, whereas later claims in the same batch complete sooner. Overall, results show that committee size is the main factor limiting claim-side scalability.

To complement the *TxBurn* and *TxClaim* measurements, we also report end-to-end transfer latency in Tables 5 and 6. This latency is measured from the source-side burn submission to the target-side finalization of the corresponding claim and therefore captures the transfer path, including burn confirmation, oracle-side commitment admission, claim submission, validator voting, batch finalization, and target-side completion.

As shown in Table 5, each value represents the end-to-end latency of a completed transfer under the fixed committee size $n = 4$. The p95 transfer latency decreases from 7596.94 ms at $b = 1$ to 6151.27 ms at $b = 20$, indicating that larger batches reduce tail completion time in the measured range. This reduction is driven by batched target-side finalization, which spreads validator coordination, aggregation, and on-chain verification across more claims. To make the within-batch effect more explicit, we also report p50 latency. The median decreases from 4637.62 ms to 4161.03 ms. This is the expected behavior of a batched system. Transfers that enter a batch earlier wait longer for the batch to fill and therefore contribute more often to the tail, whereas transfers that arrive closer to finalization complete sooner and are reflected in the median. Taken together, these results show that batching improves end-to-end transfer completion in the measured range.

For the fixed batch size $b = 20$, Table 6 shows that end-to-end p95 latency increases from 6314.73 ms at $n = 4$ to 111349.41 ms at $n = 128$. This increase is caused by the larger coordination workload introduced by larger committees. Each finalized batch must collect and process more validator responses, update the validator state, and generate the aggregated proof before the target blockchain finalization can be completed. The p50 latency also increases from 4249.76 ms to 60557.98 ms, showing that the effect is not limited to tail latency but also appears in typical completed transfers as committee-side processing scales with n. These results identify committee size as the main factor governing end-to-end latency once the batch size is fixed.

More broadly, our evaluation demonstrates that zkPACT is shaped by two main scaling factors. Batch size determines how effectively claim finalization costs are amortized, whereas committee size determines the coordination cost required before a batch can be finalized. These results highlight the need to carefully choose b and n to achieve an effective balance among efficiency, latency, and security.

Table 6
End-to-end transfer latency at fixed batch size $b = 20$.

n	E2E p50 (ms)	E2E p95 (ms)
4	4249.76	6314.73
8	4817.43	7317.09
16	9067.99	15 329.38
32	16 493.78	27 336.68
64	30 476.32	53 857.00
128	60557.98	111 349.41

7. Security and privacy analysis

This section presents the system assumptions, threat model, security and privacy analysis, and limitations of zkPACT. It builds on prior research [10,54] on attacks in blockchain interoperability, focusing on those most relevant to zkPACT.

System and cryptographic assumptions. We assume that cryptographic primitives (e.g., zk-SNARKs, hash functions, and digital signatures) are secure. We further assume that issuers act honestly [38] by verifying users during the KYC process and issuing VCs only to users who satisfy the relevant requirements. Under this assumption, successful verification of the submitted VC against the required policy conditions provides evidence that the user meets those requirements. The validator committee has the size $n \geq 3f + 1$, where at most f validators are malicious. zkPACT finalizes the batch decision with threshold $\tau = f + 1$. Under these assumptions, every finalized batch includes at least one vote from an honest validator.

Threat model. An adversary may control arbitrary users and up to f validators in the committee, including the validator currently assigned the aggregator role. Malicious users may submit arbitrary burn or claim requests, attempt to claim without satisfying the KYC policy, and reuse spent commitments or VCs. Byzantine validators may remain offline, fail to respond, submit incorrect votes, or collude to support an invalid batch. A Byzantine aggregator may deviate from the protocol during vote aggregation and batch finalization.

Quantitative collusion and Byzantine thresholds. For a committee of size n, zkPACT tolerates at most $f_{\max}(n) = \left\lfloor \frac{n-1}{3} \right\rfloor$ Byzantine validators. Accordingly, the minimum colluding set, that is, the smallest set of validators whose coordinated deviation violates the committee assumption, has size $c_{\min}(n) = f_{\max}(n) + 1$. These values define the Byzantine robustness and collusion threshold for the committee sizes considered in zkPACT; for example, given $n = 128$ committee members, our framework tolerates up to $f_{\max}(128) = 42$ Byzantine members and breaks only if more than $c_{\min}(128) = 43$ members collude.

7.1. Security analysis

Vote tampering attack. A malicious aggregator may attempt to publish a threshold-supported decision for a batch of claim requests that is not supported by validator votes collected for that batch. This attack could enable the malicious aggregator to collude with malicious users to include invalid claim requests. To mitigate this attack, zkPACT requires that the correctness of a batch be confirmed by at least $f + 1$ validators. Each validator confirms its agreement on a batch by digitally signing the vote submitted to the aggregator.

Theorem 7.1 (*Vote Tampering Resistance*)**.** *In a committee of size $n = 3f + 1$ with at most f malicious validators, an aggregator can finalize a batch of claim requests only if the batch is supported by at least $f + 1$ signed validator votes.*

Proof. Let b be a batch of claim requests supported by at most f signed validator votes, and suppose a malicious aggregator agg attempts to finalize b. To do so, agg must generate a ZKP using the Voting circuit (Algorithm 2), which enforces that validator votes for b are correctly signed (Lines 13–14) and that the number of validators supporting the batch meets a quorum of $f + 1$ (Line 21). By assumption, b is supported by at most f signed validator votes, which is insufficient to satisfy the circuit's threshold requirement. This contradiction shows that agg cannot produce a valid ZKP for b. Since b is arbitrary, we conclude that a batch of claim requests can only be finalized if it is supported by at least $f + 1$ signed validator votes. □

Consensus delay attack. Malicious validators may go offline or fail to respond to batch verification requests, delaying zkPACT consensus and reducing throughput. zkPACT mitigates this attack by ensuring that, even if up to f validators are offline or unresponsive, the remaining validators are sufficient to reach consensus on burn and claim requests, preserving protocol liveness despite reduced throughput.

Theorem 7.2 (*Consensus Delay Resistance*)**.** *An aggregator can finalize a batch of claim requests even if up to f malicious validators are offline or unresponsive.*

Proof. Let b be a batch of claim requests for which up to f malicious validators do not respond. Since our framework follows the BFT assumption, the system has $n = 3f + 1$ validators. To finalize b, the aggregator must generate a ZKP from the Voting circuit (Algorithm 2), which requires $f + 1$ signed validator votes (Line 21). Even if up to f validators are offline or unresponsive, at least $2f + 1$ validators respond to whether b is correct. Therefore, the aggregator can generate a valid ZKP and finalize b. This contradicts the assumption that the aggregator cannot finalize b. Since b is arbitrary, we conclude that the aggregator can finalize a batch of claim requests even when up to f malicious validators are offline or fail to respond. □

Sybil attack. A Sybil adversary may create or control multiple validator identities to increase its influence over voting and support an invalid batch decision. zkPACT mitigates this in two ways. First, zkPACT employs a PoS validator model [55], requiring a stake to register and to replace an existing validator, which makes Sybil creation costly. Second, zkPACT ensures that each registered validator contributes at most one signed vote per batch, preventing duplicate voting.

Theorem 7.3 (*Sybil Resistance*)**.** *For each batch of claim requests, only registered validators may have their votes counted, and each validator may vote at most once.*

Proof. Let v be a validator and agg be an aggregator. Assume that agg can generate a ZKP from the Voting circuit (Algorithm 2) for a batch b in which either v is unregistered or v's vote is counted more than once. If v's vote is counted more than once, this violates the distinct-identifier constraint (Line 2). Moreover, the circuit enforces validator membership (Line 12) and verifies the signature of each vote (Lines 13–14), ensuring that only registered validators with valid signatures can have their votes counted. Therefore, if v is unregistered or its vote is counted more than once in b, the circuit constraints are violated, a contradiction. Since v and b are arbitrary, we conclude that only registered validators may have their votes counted, and each validator may vote at most once per batch. □

Double-spending attack. Malicious users may attempt to claim tokens multiple times from a single burn commitment, either by submitting repeated claims on the same destination chain or by attempting to reuse the burn commitment on a different destination chain. zkPACT mitigates this attack by requiring that each burn commitment is consumed at most once and is valid only for the designated destination chain.

Theorem 7.4 (*Double-spending Resistance*)**.** *Malicious users cannot collude with malicious aggregators and validators to claim tokens multiple times from a single burn commitment.*

Proof. Let u be a malicious user, agg a malicious aggregator, and V_m the set of malicious validators colluding with u. Suppose u has already claimed a burn and attempts to claim it again. For the second claim to succeed, the request must pass the Claiming circuit (Algorithm 1), and agg must produce a valid ZKP from the Voting circuit (Algorithm 2). The Claiming circuit binds the public nul_{hash} to the private nul by hashing nul and asserting equality (Lines 3–4). Thus, reclaiming the same burn reuses nul_{hash}. Once nul_{hash} is recorded as spent, validators reject any later claim that reuses it.

Moreover, to prevent cross-chain reuse, the Claiming circuit recomputes the commitment c and asserts that it equals the corresponding Merkle leaf (Lines 10–11), and verifies Merkle inclusion under the published root (Line 2). Since $dest_{id}$ is part of c, claiming the burn on a different destination chain would require changing $dest_{id}$, which changes c and causes the equality check to fail. Thus, the same burn cannot be reused to claim tokens on another destination chain, and u cannot generate a valid ZKP from the Claiming circuit.

Furthermore, generating an accepted ZKP from the Voting circuit requires validator confirmation that the claim request is valid. Under our assumption, all validators in V_m would approve u's request even though u cannot generate a valid ZKP from the Claiming circuit. However, the Voting circuit requires support from at least $f + 1$ validators (Line 21), whereas at most f validators can be malicious under the BFT assumption. Therefore, V_m cannot provide enough votes. Consequently, even with collusion between u, V_m, and agg, a valid ZKP from the Voting circuit cannot be generated. Since u, V_m, and agg are arbitrary, we conclude that malicious users cannot collude with malicious aggregators and validators to claim tokens multiple times from a single burn commitment. □

Unauthorized token claim attack. Malicious users may attempt to claim tokens without a valid burn commitment or a valid VC that satisfies KYC requirements. A commitment is valid if it is included in the commitment tree and is unclaimed. A VC is valid if it is signed by a required issuer, is not revoked, and remains valid at the claim epoch. Our framework prevents this attack by requiring users to submit ZKPs from the Claiming circuit (Algorithm 1) to prove the correctness of both the burn commitment and the VC.

Theorem 7.5 (*Unauthorized Token Claim Resistance*)**.** *Malicious users can only claim tokens if the burn commitment and the VC are valid.*

Proof. Since we show in Theorem 7.4 that a claimed commitment cannot be reused, we consider in this proof only a commitment that is not included in the commitment tree. Suppose a malicious user u attempts to claim tokens using an invalid commitment c or an invalid VC vc. Specifically, c is not included in the public commitment tree, and vc is either not signed by a trusted issuer, has been revoked, or has expired. To succeed, c and vc must satisfy all constraints of the Claiming circuit (Algorithm 1). First, the circuit enforces that c must be included in the public commitment tree (Line 2). Since c is not in the tree, u cannot provide a valid Merkle path for its inclusion. Second, the circuit requires that vc is signed by the issuer's public key (Line 8), has not been revoked (Lines 6–7), and remains valid at the claim epoch (Line 9). When vc is either unsigned, revoked, or no longer valid at the claim epoch, it cannot satisfy these constraints. Since c or vc do not satisfy the circuit constraints, and since c and vc are arbitrary, we conclude that a malicious user can claim tokens only if both the burn commitment and the VC are valid. □

7.2. Privacy analysis

An attacker, which may be an external observer or a member of the validator committee, might attempt to perform the *burn-claim linking attack* that associates a claim on the destination chain with its corresponding burn on the source chain. In zkPACT, privacy is captured by two related guarantees. The first is unlinkability between the source-side burn and the target-side claim. The second is the confidentiality of the user's sensitive identity attributes. Thus, although an observer may see that a burn occurred on the source chain and that a later claim occurred on the target chain, such an observer cannot determine from protocol-visible data which specific burn commitment generated that claim, while the user's identity attributes remain hidden by the ZKP-based credential proof. Our framework mitigates this linking attack by leveraging a sufficiently large anonymity set and an unlinkability property. We first define these properties and show that zkPACT achieves a larger anonymity set size and stronger unlinkability than traditional single-chain mixers (e.g., [18]). Building on this foundation, we then analyze how our framework prevents this attack.

Definition 7.1 (*Anonymity Set Size*)**.** Let $A_{\text{zkPACT}}(t)$ denote the set of burn commitments included in the global commitment tree under the public root at time t. The anonymity set size at time t is $|A_{\text{zkPACT}}(t)|$, representing the number of burn commitments among which a user's burn becomes indistinguishable.

In traditional single-chain mixers [18], deposits and withdrawals occur on the same blockchain, and no cross-chain interactions are possible. Thus, the anonymity set at time t is limited to deposits on that chain, i.e., $|A_{\text{single}}(t)| = B(t)$, where $B(t)$ denotes the number of deposits from that blockchain currently included in the pool. In contrast, zkPACT aggregates burn commitments from multiple supported blockchains into a single global commitment tree, yielding $|A_{\text{zkPACT}}(t)| = \sum_{i=1}^{k} B_i(t)$, where k is the number of supported blockchains and $B_i(t)$ is the number of burn transactions from blockchain i included at time t. Therefore, zkPACT's cross-chain aggregation enlarges the anonymity set and increases adversarial uncertainty compared to traditional single-chain designs.

Definition 7.2 (*Unlinkability*)**.** Let $A_{\text{zkPACT}}(t)$ be the set of burn commitments included in the global commitment tree under the public root used for verification at time t. A claim is unlinkable if, given only protocol-visible data, an adversary cannot identify which element of $A_{\text{zkPACT}}(t)$ generated the claim with probability greater than $1/|A_{\text{zkPACT}}(t)|$.

Theorem 7.6 (*Burn-claim Linking Resistance*)**.** *Let tx_{br} denote a burn request on the source chain and tx_{clm} denote a claim request on the destination chain. Assuming the attacker observes only protocol-visible data, the probability of linking tx_{clm} to its originating burn tx_{br} is at most $\frac{1}{|A_{zkPACT}(t)|}$, where $A_{zkPACT}(t)$ denotes set of burn commitments included in the global commitment tree under the public root at time t.*

Proof. Let u be a target user, tx_{clm} the claim request submitted by u, and $A^{br}(t)$ the set of burn requests that an attacker could associate with tx_{clm}, including tx_{br}. The attacker's linking probability is then $1/|A^{br}(t)|$. Suppose $|A^{br}(t)| < |A_{\text{zkPACT}}(t)|$; in this case, the adversary could potentially eliminate unrelated commitment leaves from A_{zkPACT}. However, in the Claiming circuit (Algorithm 1), the leaf index and Merkle path are private inputs, and the ZKP only proves membership under the public root (Line 2). The commitment-opening values are also private, and the circuit only checks consistency by recomputing the leaf value (Lines 10–11). Therefore, the protocol-visible data does not allow the adversary to distinguish the used commitment from other members of $A_{\text{zkPACT}}(t)$. Hence, the linking probability is at most $1/|A_{\text{zkPACT}}(t)|$. □

Entropy as an uncertainty metric. In addition to the worst-case linking bound above, we use Shannon entropy [56] as a quantitative measure of the adversary's uncertainty about which commitment generated a given claim. Let q_j denote the adversary's estimated probability that commitment $j \in A_{\text{zkPACT}}(t)$ is the claim origin. The uncertainty is then $H(t) = -\sum_{j=1}^{|A_{\text{zkPACT}}(t)|} q_j \log_2 q_j$. When the adversary has no

distinguishing information, all candidates are equally likely, so $q_j = 1/|A_{\text{zkPACT}}(t)|$ and the entropy is maximized at $H(t) = \log_2 |A_{\text{zkPACT}}(t)|$. Because zkPACT aggregates burn commitments from multiple supported blockchains into a single global commitment tree, the larger anonymity set also increases the adversary's uncertainty.

7.3. Limitations and mitigation approaches

While zkPACT provides strong privacy and security guarantees, it still inherits limitations common to decentralized cross-chain protocols. A large-scale collusion or bribery attack remains a threat. If more than f validators are compromised or bribed, violating the BFT assumption, the oracle voting process may become unreliable, and false cross-chain state transitions may be finalized. In zkPACT, stake locking, slashing, and reputation effects increase the cost of such collusion. Moreover, this risk can be further reduced through greater validator decentralization and through future mechanisms such as committee rotation and dynamic scoring that limit long-term control by a fixed subset of validators. Even with batching and enforced delays, an adversary with global monitoring capabilities may still detect patterns between burns and token claims, allowing educated guesses about transaction links. One possible mitigation is to use a Verifiable Delay Function (VDF), which introduces an unpredictable delay between a user's claim request and its finalization, making correlations more difficult.

zkPACT also assumes an honest issuer that retains the off-chain identity mapping for accountability. If the issuer becomes malicious, user identities may be revealed. This reflects the trade-off between privacy and regulatory accountability in compliant systems. A possible mitigation is threshold-based identity recovery, where disclosure requires multiple independent issuers. During periods of low network activity, participation in deposits and token claims may decline, shrinking the anonymity set. A smaller anonymity set weakens privacy guarantees by making it easier to correlate individual user actions. This challenge can be mitigated through adaptive anonymity scaling and dummy transactions that help preserve unlinkability even when real activity is limited.

Moreover, although zkPACT employs a combined stake-and-reputation scoring system for validator selection, a fixed weighting between these components may be suboptimal under changing network conditions. For example, in high-load scenarios, responsiveness may deserve more weight than stake alone. A dynamic scoring mechanism that adjusts the influence of stake and reputation based on validator performance and network activity could therefore help maintaining a responsive validator set as the protocol evolves.

Finally, in a realistic multi-chain operating environment, zkPACT depends on source-chain-specific finality handling before oracle-approved burns are admitted into the shared commitment tree. Because participating chains may use different confirmation and finality models, the oracle layer should apply confirmation thresholds tailored to each source chain, while pre-confirmation can improve responsiveness without replacing finality for safety. This is equally important for safe insertion of burn commitments into the global tree, since admitting an event too early may expose the system to the rollback of non-final source-chain blocks or other exceptional failures. For this reason, production deployment should delay commitment admission until the relevant source-chain finality condition is satisfied and should define a recovery procedure, such as committee-coordinated rollback of the affected root, for cases involving non-final source-chain rollback, validator faults, or network inconsistencies. On the destination side, the current verifier stack can be deployed directly on EVM-compatible runtimes with efficient pairing-based zk-SNARK support, while other environments may require adapting the verifier and backend to the execution model of the target chain.

8. Conclusions and future work

This study addresses the fragmentation of blockchain ecosystems, which hinders seamless cross-platform interaction and poses challenges for systems that rely on them, such as BISs. zkPACT bridges these ecosystems with a privacy-preserving cross-chain token transfer framework that integrates ZKPs, verifiable credentials, and decentralized oracle networks.

This design enforces KYC compliance through verifiable credentials, preserves anonymity under legitimate use, and enables accountability in cases of misuse. We leverage off-chain proof aggregation and batch verification to improve scalability and reduce computational costs. Our framework achieves up to 95% lower on-chain gas usage and up to 94% lower off-chain memory consumption than a non-batching approach. The incentive and slashing mechanisms promote honest participation by adjusting penalties and rewards based on validator behavior and past reliability. Looking ahead, we plan to explore enhanced rollback mechanisms for resilient state synchronization and investigate preconfirmation mechanisms to reduce cross-chain latency and improve user experience.

CRediT authorship contribution statement

Elmira Ebrahimi: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Anh-Tu Hoang:** Writing – review & editing. **Dominik Kaaser:** Writing – review & editing. **Michael Sober:** Writing – review & editing. **Juan M. Tirado:** Writing – review & editing. **Stefan Schulte:** Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Elmira Ebrahimi reports financial support was provided by Christian Doppler Research Association. Anh-Tu Hoang reports financial support was provided by Christian Doppler Research Association. Dominik Kaaser reports was provided by Christian Doppler Research Association. Michael Sober reports financial support was provided by Christian Doppler Research Association. Stefan Schulte reports financial support was provided by Christian Doppler Research Association. Juan M. Tirado reports a relationship with Bitpanda GmbH that includes: employment. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology, and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] H. Guo, X. Yu, A survey on blockchain technology and its security, Blockchain: Res. Appl. 3 (2) (2022) 100067.

[2] D. Berdik, S. Otoum, N. Schmidt, D. Porter, Y. Jararweh, A survey on blockchain for information systems management and security, Inf. Process. Manage. 58 (1) (2021) 102397.

[3] A. Hajian, V.R. Prybutok, H.-C. Chang, An empirical study for blockchain-based information sharing systems in electronic health records: A mediation perspective, Comput. Hum. Behav. 138 (2023) 107471.

[4] R.K. Singh, R. Mishra, S. Gupta, A.A. Mukherjee, Blockchain applications for secured and resilient supply chains: A systematic literature review and future research agenda, Comput. Ind. Eng. 175 (2023) 108854.

[5] S. Mathur, A. Kalla, G. Gür, M.K. Bohra, M. Liyanage, A survey on role of blockchain for IoT: Applications and technical aspects, Comput. Netw. 227 (2023) 109726.

[6] W. Li, Z. Liu, J. Chen, Z. Liu, Q. He, Towards blockchain interoperability: A comprehensive survey on cross-chain solutions, Blockchain: Res. Appl. 6 (3) (2025) 100286.

[7] G. Wang, Q. Wang, S. Chen, Exploring blockchains interoperability: A systematic survey, ACM Comput. Surv. 55 (13s) (2023) 290:1–290:38.

[8] K. Ren, N.-M. Ho, D. Loghin, T.-T. Nguyen, B.C. Ooi, Q.-T. Ta, F. Zhu, Interoperability in blockchain: A survey, IEEE Trans. Knowl. Data Eng. 35 (12) (2023) 12750–12769.

[9] R.V. Tate, S.B. Mane, A review on current state of art of a blockchain interoperability, ICT Express 12 (2) (2026) 444–458.

[10] A. Augusto, R. Belchior, M. Correia, A. Vasconcelos, L. Zhang, T. Hardjono, SoK: Security and privacy of blockchain interoperability, in: IEEE Symposium on Security and Privacy, IEEE, 2024, pp. 3840–3865.

[11] Y. Guo, M. Xu, X. Cheng, D. Yu, W. Qiu, G. Qu, W. Wang, M. Song, zkCross: A novel architecture for cross-chain privacy-preserving auditing, in: USENIX Security Symposium, USENIX Association, 2024, pp. 6219–6235.

[12] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, D. Song, zkBridge: Trustless cross-chain bridges made practical, in: ACM SIGSAC Conference on Computer and Communications Security, ACM, 2022, pp. 3003–3017.

[13] B. Zhang, J. Xu, X. Wang, Z. Zhao, S. Chen, X. Zhang, Research on the construction of grain food multi-chain blockchain based on zero-knowledge proof, Foods 12 (8) (2023) 1600.

[14] X. Wu, T. Zhang, L. Chen, Z. Wang, Privacy-preserving cross-chain asset transfers using notary schemes and zero-knowledge proofs, Clust. Comput. 28 (2025) 387.

[15] X. Xiong, M. Huth, W. Knottenbelt, REGKYC: Supporting privacy and compliance enforcement for KYC in blockchains, in: 2025 IEEE International Conference on Blockchain and Cryptocurrency, IEEE, 2025, pp. 1–5.

[16] J. Heiss, J. Eberhardt, S. Tai, From oracles to trustworthy data on-chaining systems, in: IEEE International Conference on Blockchain, IEEE, 2019, pp. 496–503.

[17] E. Ben-Sasson, A. Chiesa, E. Tromer, M. Virza, Succinct non-interactive zero knowledge for a von Neumann architecture, in: 23rd USENIX Security Symposium, USENIX Association, 2014, pp. 781–796.

[18] D. Morales, I. Agudo, J. Lopez, Zero-knowledge bitcoin mixer with reversible unlinkability, Blockchain: Res. Appl. (2025) 100323.

[19] T. Lavaur, J. Detchart, J. Lacan, C.P.C. Chanel, Modular zk-Rollup on-demand, J. Netw. Comput. Appl. 217 (2023) 103678.

[20] C. Mazzocca, A. Acar, S. Uluagac, R. Montanari, P. Bellavista, M. Conti, A survey on decentralized identifiers and verifiable credentials, IEEE Commun. Surv. & Tutorials 27 (6) (2025) 3641–3671.

[21] M. Campbell-Verduyn, Bitcoin, crypto-coins, and global anti-money laundering governance, Crime, Law Soc. Chang. 69 (2018) 283–305.

[22] R. Belchior, L. Riley, T. Hardjono, A. Vasconcelos, M. Correia, Do you need a distributed ledger technology interoperability solution? Distrib. Ledger Technol.: Res. Pr. 2 (1) (2023) 1–37.

[23] M. Sober, G. Scaffino, S. Schulte, Cross-blockchain communication using oracles with an off-chain aggregation mechanism based on zk-SNARKs, Distrib. Ledger Technol.: Res. Pr. 3 (4) (2024) 1–24.

[24] M. Herlihy, Atomic cross-chain swaps, in: ACM Symposium on Principles of Distributed Computing, ACM, 2018, pp. 245–254.

[25] H. Yuan, S. Fei, Z. Yan, Technologies of blockchain interoperability: a survey, Digit. Commun. Netw. 11 (1) (2025) 210–224.

[26] P. Frauenthaler, M. Sigwart, C. Spanring, M. Sober, S. Schulte, ETH relay: A cost-efficient relay for ethereum-based blockchains, in: IEEE International Conference on Blockchain, IEEE, 2020, pp. 204–213.

[27] R. Belchior, D. Dimov, Z. Karadjov, J. Pfannschmidt, A. Vasconcelos, M. Correia, Harmonia: Securing cross-chain applications using zero-knowledge proofs, 2026, http://dx.doi.org/10.36227/techrxiv.170327806.66007684/v5, Preprint.

[28] M.S. Peelam, B.K. Chaurasia, A.K. Sharma, V. Chamola, B. Sikdar, Unlocking the potential of interconnected blockchains: A comprehensive study of cosmos blockchain interoperability, IEEE Access 12 (2024) 171753–171776.

[29] Y. Cao, J. Cao, D. Bai, L. Wen, Y. Liu, R. Li, Map the blockchain world: A trustless and scalable blockchain interoperability protocol for cross-chain applications, in: Web Conference, ACM, 2025, pp. 717–726.

[30] Y. Ren, Z. Lv, N.N. Xiong, J. Wang, HCNCT: A cross-chain interaction scheme for the blockchain-based metaverse, ACM Trans. Multimed. Comput. Commun. Appl. 20 (7) (2024) 1–23.

[31] L.T. Thibault, T. Sarry, A.S. Hafid, Blockchain scaling using rollups: A comprehensive survey, IEEE Access 10 (2022) 93039–93054.

[32] D.L. Fekete, A. Kiss, Trust-minimized optimistic cross-rollup arbitrary message bridge, J. Netw. Comput. Appl. 221 (2024) 103771.

[33] O. Goldreich, Y. Oren, Definitions and properties of zero-knowledge proof systems, J. Cryptology 7 (1) (1994) 1–32.

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

[35] A.M. Pinto, An introduction to the use of zk-SNARKs in blockchains, in: 1st International Conference on Mathematical Research for Blockchain Economy, Springer, 2020, pp. 233–249.

[36] J. Groth, On the size of pairing-based non-interactive arguments, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2016, pp. 305–326.

[37] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, L. van der Maaten, CrypTen: Secure multi-party computation meets machine learning, Adv. Neural Inf. Process. Syst. 34 (2021) 4961–4973.

[38] M. Rosenberg, J. White, C. Garman, I. Miers, zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure, in: IEEE Symposium on Security and Privacy, IEEE, 2023, pp. 790–808.

[39] W. Huang, Y. Deng, J. Feng, G. Han, W. Zhang, Securing hashed timelock cross-chain protocol with trusted middleman in blockchain networks, Comput. Stand. Interfaces 97 (2026) 104129.

[40] T. Li, P. Niu, Y. Wang, S. Zeng, X. Wang, W. Susilo, HT2REP: A fair cross-chain atomic exchange protocol under UC framework based on HTLCs and TRE, Comput. Stand. Interfaces 89 (2024) 103834.

[41] A. Pathak, I. Al-Anbagi, H. Hamilton, SATI: Sidechain-based access control & trust mechanism for IoT networks, IEEE Trans. Netw. Serv. Manag. 21 (5) (2024) 5888–5903.

[42] Y. Guo, H. Zhu, M. Xu, X. Cheng, B. Xiao, xRWA: A cross-chain framework for interoperability of real-world assets, High-Confidence Comput. (2026) 100380.

[43] S. Xu, L. Zhang, L. Wang, M.J. Mihaljević, S. Zhang, W. Shao, Q. Wang, Relay network-based cross-chain data interaction protocol with integrity audit, Comput. Electr. Eng. 117 (2024) 109262.

[44] Y. Sun, L. Yi, L. Duan, W. Wang, A decentralized cross-chain service protocol based on notary schemes and hash-locking, in: IEEE International Conference on Services Computing, IEEE, 2022, pp. 152–157.

[45] D. Morháč, K. Košt'ál, V. Valaštín, I. Kotuliak, Unispell: universal adapter for interoperability in Polkadot paraverse, Clust. Comput. 28 (5) (2025) 311.

[46] K. Chang, O. Seneviratne, Zero-knowledge proof framework for identity verification and interoperable payments on the decentralized web, ACM Trans. Web (2026).

[47] M. Sober, M. Levonyak, S. Schulte, Efficient cross-blockchain token transfers with rollback support, in: IEEE International Conference on Decentralized Applications and Infrastructures, IEEE, 2024, pp. 9–18.

[48] X. Hu, X. Chen, Z. Dong, Y. Sun, Y. Guo, B. Fang, J. Qi, Optimized cross-chain transactions with aggregated zero-knowledge proofs: Enhancing efficiency and security, IEEE Internet Things J. 12 (9) (2025) 11495–11510.

[49] L. Zavolokina, I. Bauer-Hänsel, J. Hacker, G. Schwabe, Organizing for value creation in blockchain information systems, Inf. Organ. 34 (3) (2024) 100522.

[50] A.-T. Hoang, C.U. Ileri, W. Sanders, S. Schulte, zkSSI: A zero-knowledge-based self-sovereign identity framework, in: IEEE International Conference on Blockchain, IEEE, 2024, pp. 276–285.

[51] G. Laatikainen, M. Li, P. Abrahamsson, A system-based view of blockchain governance, Inf. Softw. Technol. 157 (2023) 107149.

[52] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, T. Tiessen, MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2016, pp. 191–219.

[53] L. Breidenbach, C. Cachin, A. Coventry, Y. Ji, K. Karantias, P. Schindler, C. Stathakopoulou, A. Topliceanu, Chainlink Offchain Reporting Protocol 3.0, 2025, (Accessed 10 March 2026), https://research.chain.link/ocr3.pdf.

[54] T. Haugum, B. Hoff, M. Alsadi, J. Li, Security and privacy challenges in blockchain interoperability-a multivocal literature review, in: 26th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2022, pp. 347–356.

[55] A. Gangwal, H.R. Gangavalli, A. Thirupathi, A survey of layer-two blockchain protocols, J. Netw. Comput. Appl. 209 (2023) 103539.

[56] C. Diaz, S. Seys, J. Claessens, B. Preneel, Towards measuring anonymity, in: International Workshop on Privacy Enhancing Technologies, Springer, 2002, pp. 54–68.