A Prototype-Based Study of Zero-Knowledge Proof Verification for Privacy-Preserving Blockchain Interoperability

Chilume O. Gabriel, Hlomani B. Hlomani, and Kabo Nkabiti
Department of Computing and Informatics
Botswana International University of Science and Technology
Palapye, Botswana
{gabrielc, hlomanihb, nkabitik}@biust.ac.bw

***Abstract*—Blockchain networks need to exchange messages and assets across independent systems, but cross-chain verification can expose private validation data to relayers, bridge logic, validators, or destination-chain components. This paper presents a prototype-based zero-knowledge verification layer for privacy-preserving blockchain interoperability. The prototype uses Circom and SnarkJS to generate Groth16 proofs, verifies those proofs in Rust using arkworks BN254, and maps the result into a Substrate-style interoperability decision model. The work addresses a practical implementation gap between common zero-knowledge proof tools and Rust-based blockchain interoperability environments. Private values stay off-chain, while only the proof, public commitment, verification metadata, and final decision are passed to the runtime-facing layer. The prototype includes valid-proof acceptance, tampered-input rejection, runtime-compatible verification records, and a simulated interoperability decision layer. Experimental results show proof generation at 199 ms, SnarkJS valid-proof verification at 161 ms, tampered-input rejection at 160 ms, and Rust verifier execution at 340 ms. These results show that a SnarkJS-generated Groth16 proof can be verified in Rust and used to control whether a simulated cross-chain action is accepted or rejected. The current scope does not include a full FRAME pallet or live Cross-Consensus Messaging (XCM) dispatch. In Polkadot, XCM is the message format used to send instructions between different chains. The prototype is not intended to replace Polkadot's existing parachain auditing mechanisms. Instead, it explores a complementary privacy-preserving verification path for selected interoperability conditions where private inputs should not be exposed. The prototype provides a repeatable technical path for building privacy-preserving verification in Polkadot/Substrate-style interoperability workflows.**

***Index Terms*—Blockchain interoperability, zero-knowledge proofs, zk-SNARKs, Groth16, Polkadot, Substrate, XCM, Circom, SnarkJS, Rust verification, arkworks BN254.**

I. Introduction

Blockchain technology is used in financial systems, smart contracts, digital identity, supply-chain systems, and other distributed applications because it supports decentralized and tamper-resistant record keeping [1], [2], [13], [14]. However, many blockchain networks still operate as separate systems. There is no single blockchain platform or common protocol that covers every use case. This limits practical use in multi-chain environments where different chains must exchange messages, assets, and state information.

Blockchain interoperability addresses this problem by allowing independent chains to communicate. Polkadot supports cross-consensus communication through Cross-Consensus Messaging (XCM), while Cosmos IBC uses a light-client-based model for communication among sovereign chains [2]–[4], [15], [16]. Polkadot also has existing parachain validity auditing mechanisms, including ELVES, which help check parachain blocks under Byzantine assumptions. This paper does not replace those mechanisms. Instead, it focuses on a smaller and more specific question: how private conditions can be proven without exposing the private data behind them. Bridges and relay systems also support cross-chain communication, but they often introduce new trust and privacy risks [5], [8]. In some workflows, validation data may become visible to relayers, bridge validators, destination-chain logic, or other intermediaries. This is a problem where the receiving chain only needs to know that a condition is true, not the private data behind that condition.

Zero-Knowledge Proofs (ZKPs) provide a way to prove that a statement is true without revealing the private data used to produce the proof [6], [7], [17], [18]. In cross-chain verification, this means that a source-side condition can be proven while balances, identity values, nullifiers, or other sensitive inputs remain hidden. This paper studies how such a proof can be generated with common ZKP tools and then verified in a Rust-based environment that is closer to Polkadot/Substrate development.

This paper presents a prototype-based implementation and feasibility study. The prototype does not claim to be a production Polkadot deployment. It does not implement a full FRAME pallet, and it does not execute live XCM dispatch between running parachains. Its purpose is to provide implementation evidence for a repeatable path from private proof generation to Rust-based Groth16 verification and Substrate-style interoperability decision control.

The novelty of this work is not the invention of a new proving system or a new cross-chain protocol. Instead, the contribution is a repeatable implementation workflow that connects Circom/SnarkJS Groth16 proof generation to Rust arkworks BN254 verification and then maps the verification result into a Substrate-style interoperability decision layer. This addresses a practical gap between ZKP proof generation

tools and Rust-based blockchain interoperability environments. The prototype provides implementation evidence, accept/reject validation, timing results, and repeatable artifacts that can support future integration into a FRAME pallet.

The main contributions are:

- a repeatable workflow from Circom/SnarkJS Groth16 proof generation to Rust arkworks BN254 verification;
- a technical mapping from Rust verification output to a Substrate-style verifier adapter and runtime-compatible verification record;
- an accept/reject validation path showing that a valid proof is accepted and a tampered public input is rejected;
- a comparative analysis against Polkadot XCM, Cosmos IBC, traditional bridges, and zk-SNARK-based verification models;
- measured prototype timings for proof generation, SnarkJS verification, tampered-input rejection, and Rust verifier execution; and
- a repeatability package containing selected proof artifacts, scripts, timing output, documentation, and demo commands.

II. Related Work and Comparative Gap

Existing interoperability approaches include relay chains, bridges, notary schemes, light-client protocols, and cross-chain message passing [1], [2], [23]. Polkadot uses a relay-chain and parachain model, while XCM provides a message format for communication across consensus systems [3], [4]. Cosmos IBC relies on light-client verification among sovereign chains [2]. Polkadot's ELVES mechanism is also relevant to this work. ELVES is an efficient parachain block-auditing layer used in Polkadot and Kusama. It uses availability distribution, randomly selected auditors, approvals, and escalation to check parachain validity under Byzantine assumptions. This paper does not compete with ELVES and does not claim to replace Polkadot's existing auditing design. The focus here is narrower. It studies whether a zero-knowledge proof can be generated off-chain, verified in Rust, and then used as an additional privacy-preserving decision input for selected interoperability workflows.

Cross-chain bridges have been widely used, but they are often limited by trust assumptions around bridge validators, relayers, multisignature groups, or external verification services [5], [8]. A bridge can move information across networks, but the bridge logic may become a high-value target. It may also require validation data to be visible outside the source environment. This creates a need for verification methods that can prove correctness without revealing the private inputs.

ZKPs have been applied to privacy-preserving blockchain systems, including value exchange, identity authentication, healthcare access control, petitions, and distributed learning [6], [7], [9]. VeCroToken, for example, uses zk-SNARKs for privacy-preserving value exchange in consortium blockchains [7]. Other works study performance and benchmarking of Groth16 and related proof systems [10], [11], [19]–[21]. However, many ZKP-based blockchain works focus on Ethereum-like settings, Hyperledger Fabric, or specific application domains. Less practical evidence is available for a repeatable path from Circom/SnarkJS proof generation to Rust Groth16 verification and Substrate-style interoperability control.

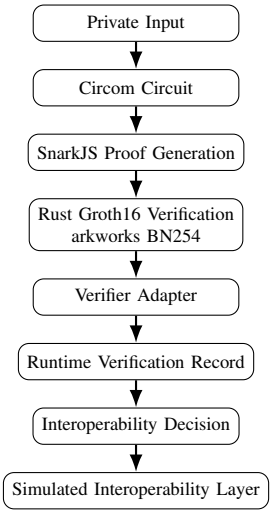

Fig. 1. End-to-end prototype flow from private input to a simulated interoperability decision layer.

Table I compares the proposed prototype with common interoperability approaches. The prototype does not replace XCM, IBC, bridges, or Polkadot's existing parachain auditing mechanisms. It adds a privacy-focused verification path for cases where private validation data should remain off-chain.

III. Prototype Design

The prototype separates private computation from public verification. Private inputs are used to generate a witness and proof outside the runtime-facing layer. The runtime-facing layer receives only the proof, a public commitment or public input, verification metadata, and the decision result. Fig. 1 summarizes this flow.

A. Input Visibility Model

Table II shows which values remain private and which values are visible to the runtime-facing layer. The important design choice is that private transaction values are not passed into the interoperability decision layer. The decision layer only receives data needed to verify and record the proof outcome.

B. Verification and Decision Model

Let π denote the Groth16 proof, vk the verification key, and x the public input or commitment. The verifier evaluates the proof as follows:

$$V(vk, x, \pi) \rightarrow \{true, false\}. \tag{1}$$

The verification output is mapped to an interoperability decision:

$$D = \begin{cases} Accept, & \text{if } V(vk, x, \pi) = true, \\ Reject, & \text{if } V(vk, x, \pi) = false. \end{cases} \tag{2}$$

TABLE I
COMPARISON WITH EXISTING BLOCKCHAIN INTEROPERABILITY APPROACHES

Approach	Verification method	Privacy support	Trust assumption	ZKP	Main limitation
Polkadot XCM	Relay-chain and parachain messaging	Depends on message content	Relay-chain and parachain security	No	Does not directly verify private proofs.
Polkadot ELVES	Availability, auditing, approvals, and escalation	Focuses on parachain validity, not private-input hiding	Polkadot validator and auditing assumptions	No	Not designed as a private-condition proof layer.
Cosmos IBC	Light-client verification	Limited by packet data	Correct light-client updates	No	Needs correct client maintenance.
Traditional bridges	Validators, relayers, or multisig groups	Often limited	Bridge operators or validators	Usually no	Bridge compromise and trust risk.
zk-SNARK verification	Cryptographic proof checking	Strong when inputs stay off-chain	Proof system and verifier correctness	Yes	Adds proof cost and integration work.
This prototype	Circom/SnarkJS proof and Rust Groth16 check	Private inputs remain off-chain	Verifier, proof artifacts, and Groth16 setup	Yes	Decision layer is simulated; FRAME pallet not complete.

TABLE II
INPUT VISIBILITY MODEL

Input	Example	Visibility
Sender balance	Balance used in the proof	Private
Transfer amount	Amount or condition value	Private
Secret/nullifier	Replay-prevention value	Private
Commitment	Hash linked to private value	Public
Destination chain	Target chain identifier	Public
Decision result	Accepted or rejected	Public

A valid proof allows the simulated cross-chain action to continue. A failed proof blocks it. This creates a traceable verification decision that can be stored, reviewed, and later adapted to runtime logic.

IV. TECHNICAL IMPLEMENTATION

This section expands the implementation details to show how the prototype was built and where the main technical connection occurs.

A. Circuit Design

The Circom circuit encodes the condition that must be proven. The witness is created from private inputs and a public commitment. The private values are not exported to the runtime-facing layer. The circuit output is a public signal that can be checked against the proof and verification key. In the current prototype, the circuit is intentionally small so that the end-to-end verification path can be tested without hiding the implementation behind a large application-specific circuit.

B. Proof Artifact Generation

SnarkJS is used to compile the circuit, generate the witness, and produce a Groth16 proof. The produced artifacts include the proof, public input, and verification key. The proof is first checked with SnarkJS before it is passed into the Rust verifier. This local verification step confirms that the proof and public input match the verification key before cross-tool verification is attempted.

C. SnarkJS-to-Rust Conversion

The central engineering issue is that SnarkJS and Rust arkworks represent proof data differently. The SnarkJS proof and public input files are JSON-based. The Rust verifier must parse these values, convert field elements into BN254-compatible representations, and prepare the verification key and proof structures expected by arkworks. This conversion step is important because a formatting or ordering error can cause a valid proof to fail in Rust.

D. Rust Groth16 Verification

The Rust verifier uses arkworks BN254 to verify the SnarkJS-generated Groth16 proof. It loads the verification key, proof, and public input, converts them into the correct structures, and runs Groth16 verification. The verifier returns a Boolean result. A successful result means that the proof, public input, and verification key are consistent. A failed result means that the proof context is invalid or has been changed.

E. Verifier Adapter and Runtime Record

The verifier result is passed into a Substrate-style adapter. The adapter is not a full pallet. It models how a runtime-facing component can receive a proof result and prepare a verification record. The record contains the proof hash, public commitment, verifier identifier, timestamp, decision, and reason message. This makes the result traceable and suitable for later movement into FRAME storage and events.

F. Interoperability Decision Logic

The decision logic maps a valid proof to *Accept* and an invalid proof to *Reject*. In the current scope, this decision controls a simulated interoperability action. The simulated action stands in for a later XCM dispatch. The purpose is to test the control point: the cross-chain action should only proceed after proof verification succeeds.

Algorithm 1 summarizes the verification flow used by the prototype. The algorithm starts by loading the Groth16 proof, public input, and verification key produced by the Circom/SnarkJS workflow. The proof fields are then converted

Algorithm 1 ZKP-Based Interoperability Verification

Require: Proof π, verification key vk, public commitment x, destination chain d
Ensure: $Accept$ or $Reject$
Load proof π and public input x
Load verification key vk
Convert SnarkJS proof fields into BN254 Rust structures
Run Groth16 verification $V(vk, x, \pi)$
if $V(vk, x, \pi) = true$ **then**
    Create runtime verification record
    Mark decision as $Accept$
    Allow simulated cross-chain action to destination d
else
    Create rejection record
    Mark decision as $Reject$
    Block simulated cross-chain action
end if
return decision

TABLE III
IMPLEMENTATION COMPONENTS

Component	Implemented Role
Circom circuit	Encodes the condition to be proven and separates private inputs from public signals.
SnarkJS	Generates witness, Groth16 proof artifacts, and local proof verification.
Rust/arkworks BN254	Verifies SnarkJS-generated Groth16 proof in Rust.
Conversion layer	Parses JSON artifacts and maps field values into BN254-compatible Rust structures.
Verifier adapter	Connects proof result to Substrate-style logic.
Runtime record	Stores proof hash, commitment, timestamp, verifier, decision, and reason.
Decision layer	Allows or blocks a simulated interoperability action.

into BN254-compatible Rust structures before verification is performed using arkworks. The verification result is not treated as an isolated cryptographic output. It is mapped into a runtime-facing decision record. If verification succeeds, the simulated interoperability action is allowed. If verification fails, the action is blocked. This makes the algorithm the control point between private proof generation and Substrate-style interoperability logic.

The algorithm is intentionally written as a prototype-level control flow rather than a full FRAME pallet procedure. In a production implementation, steps 6 to 12 would be implemented through pallet storage, events, extrinsics, and weight-benchmarked dispatch logic. In the current work, these steps are represented through a verifier adapter, runtime-compatible record, and simulated interoperability decision layer. This keeps the prototype repeatable while showing clearly how the verification result controls the cross-chain decision.

TABLE IV
ACCEPT/REJECT VALIDATION RESULTS

Case	Action	Result
Valid proof	Original proof and public input used.	Accepted
Tampered input	Public input changed after proof generation.	Rejected
Privacy check	Private values not sent to runtime layer.	Preserved
Rust check	SnarkJS proof verified in Rust.	Passed
Decision control	Valid proof allowed action; invalid proof blocked it.	Confirmed

TABLE V
MEASURED PROTOTYPE TIMING RESULTS

Operation	Average	Result
Proof generation	199 ms	Completed
SnarkJS valid-proof verification	161 ms	Accepted
Tampered-input rejection	160 ms	Rejected
Rust Groth16 verification	340 ms	Successful

V. EVALUATION RESULTS

The evaluation focused on correctness, timing, and repeatability. It asked whether the prototype could generate a proof, verify it in Rust, reject a tampered input, and use the result to control a simulated interoperability decision.

A. Accept/Reject Validation

Two main cases were tested. The first used the original proof, public input, and verification key produced by the Circom/SnarkJS workflow. This proof was accepted. The second used the same proof, but one public input value was deliberately changed. This tampered context was rejected. Table IV summarizes the results.

The rejection case is important because it shows that the proof cannot simply be reused with a different public statement. If the public input or commitment is changed after proof generation, verification fails and the simulated cross-chain action is blocked.

B. Timing Results and Interpretation

Timing tests were repeated ten times. The first run was excluded as a warm-up because it included process startup, file loading, and runtime initialization. The averages from the remaining runs are shown in Table V and Fig. 2.

The Rust verification time of 340 ms includes command execution, file loading, proof parsing, public input conversion, verification-key loading, and Groth16 verification. It should therefore be interpreted as end-to-end verifier execution time rather than pure cryptographic verification time. This distinction is important because runtime integration would require separating cryptographic verification cost from surrounding I/O and conversion overhead.

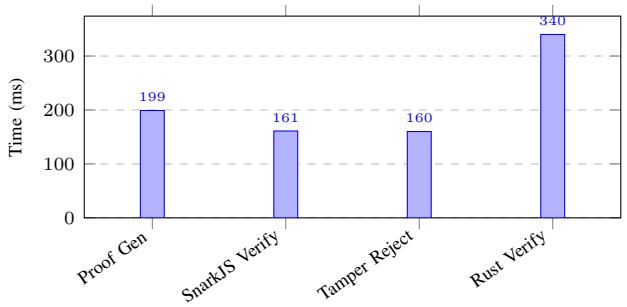

Fig. 2. Measured proof generation and verification time. Each test was repeated ten times, with the first run excluded as warm-up.

TABLE VI
RESULT INTERPRETATION SUMMARY

Metric	Result	Meaning
Proof generation	199 ms	Off-chain proof creation cost
SnarkJS verification	161 ms	Baseline toolchain check
Tampered rejection	160 ms	Rejection of changed public input
Rust verification	340 ms	Feasibility of Rust-based verification
Private input exposure	None	Private values stay off-chain
Live XCM dispatch	Not done	Current prototype boundary

C. Comparison with Traditional Verification

Traditional cross-chain systems may avoid ZKP-specific proof processing because they do not generate or verify Groth16 proofs. However, they still perform other verification steps, such as signature checking, bridge consensus, light-client verification, relayer processing, or validator approval. The difference is that sensitive validation data may be visible to intermediaries.

The proposed prototype adds proof-processing cost, but private inputs remain off-chain. Only the proof, commitment, metadata, and verification decision are exposed. This creates a clear trade-off: the ZKP layer adds measurable computation, but it improves privacy and gives the interoperability layer a proof-based control point.

D. Repeatability Procedure

The demonstration is designed to be repeatable. It starts with proof artifacts generated from the Circom/SnarkJS workflow. The reviewer first runs the valid-proof verification command. The expected result is acceptance. The reviewer then runs the tampered-input test, where one public input value has been changed. The expected result is rejection. This confirms both the positive and negative paths.

The repeatability package includes the README, Circom files, selected proof artifacts, Rust verifier, adapter/model files, timing results, and demo commands. The package also states the prototype boundary: the interoperability decision is simulated and not yet a live XCM dispatch.

TABLE VII
SECURITY AND FEASIBILITY SUMMARY

Aspect	Prototype Evidence
Privacy	Private inputs are not submitted to the runtime-facing layer.
Correctness	Valid proof accepted and tampered public input rejected.
Rust readiness	SnarkJS-generated proof verified using arkworks BN254.
Decision control	Verification result controls allow/block behaviour.
Performance	Measured proof and verification steps complete below one second.
Repeatability	Scripts, artifacts, timing results, and demo guide are packaged.
Boundary	Interoperability decision is simulated, not live XCM dispatch.

VI. SECURITY AND FEASIBILITY DISCUSSION

The prototype provides implementation evidence, but it should be interpreted within its boundary. It does not replace the Polkadot runtime, and it does not claim that live XCM dispatch has already been executed. Its purpose is to show that private proof generation, Rust verification, accept/reject validation, and Substrate-style decision control can work together.

A. Privacy Benefit

The main privacy benefit is that private inputs never cross the runtime boundary. In a traditional cross-chain flow, validation may require transaction details, state values, account balances, identity values, or message context to be visible to intermediaries. In this prototype, the verifier receives a proof and public commitment. If the proof is valid, the condition is accepted; otherwise, the action is rejected.

B. Tamper Resistance

The accept/reject test provides evidence that the verifier does not accept changed public statements. A valid proof and correct public input were accepted. When one public input value was deliberately changed, the same proof was rejected. In interoperability settings, this matters because an attacker should not be able to change the commitment, target chain data, or verification context after proof generation.

C. Computation Cost

The main cost is additional proof processing. Traditional bridge systems may avoid Groth16 proof verification, but they usually pay for other validation steps. In the prototype, proof generation happens off-chain and the measured verification steps complete below one second in the test environment. The timing results support feasibility for prototype-level control, but they do not replace runtime weight benchmarking.

VII. Limitations

The current prototype does not claim production readiness. It does not implement a full FRAME pallet, does not benchmark extrinsic weight, and does not execute live XCM dispatch between running parachains. These limitations define the prototype boundary. The contribution is the verified technical path from private proof generation to Rust-based Groth16 verification and Substrate-style decision control. The next engineering stage is to move this path into a benchmarked FRAME pallet with storage, events, extrinsics, and live XCM testing.

A second limitation is that the Rust timing value includes file loading and conversion work. A production pallet would need a different benchmark method that separates cryptographic verification cost from I/O and setup cost. A third limitation is that the circuit used in this prototype is intentionally small. Larger circuits may increase proof generation time and may require further optimization.

A. Relationship to ELVES

Polkadot already uses ELVES as part of its parachain validity auditing design. ELVES addresses an important system problem: how parachain blocks can be checked efficiently under Byzantine assumptions. The present work addresses a different and smaller problem. It asks whether selected private conditions can be proven off-chain and verified in a Rust/Substrate-style environment without revealing the private inputs.

The prototype should therefore be viewed as a possible complementary privacy layer, not as a replacement for ELVES. A future version may study a hybrid approach where normal parachain validity continues to be handled by existing Polkadot mechanisms, while specialised zero-knowledge proofs are used only where private data should stay hidden. This may be useful for cases such as private eligibility checks, identity proofs, confidential approvals, or other workflows where the system needs proof of correctness but not the underlying private record.

VIII. Conclusion

This paper presented a prototype-based implementation and feasibility study of zero-knowledge proof verification for privacy-preserving blockchain interoperability. The prototype generated Groth16 proofs with Circom and SnarkJS, verified a SnarkJS-generated proof in Rust using arkworks BN254, mapped the result into a Substrate-style verifier adapter, and used the verification decision to allow or block a simulated interoperability action.

The prototype accepted a valid proof and rejected a deliberately modified public input. Timing tests showed 199 ms for proof generation, 161 ms for SnarkJS valid-proof verification, 160 ms for tampered-input rejection, and 340 ms for Rust verifier execution. The results provide implementation evidence for a repeatable path from private proof generation to Rust-based verification and Substrate-style decision control. The current scope is limited to a simulated interoperability decision layer, but the same path can support later development of a full FRAME pallet and live XCM testing.

Acknowledgment

The authors thank their institution and the open-source communities that maintain Circom, SnarkJS, arkworks, and the Polkadot SDK.

Data Availability

The data and selected artifacts used to support the findings of this study are available from the corresponding author upon reasonable request. The repeatability package includes selected proof artifacts, scripts, timing results, documentation, and demo commands.

Declaration of Generative AI and AI-Assisted Technologies

During the preparation of this work, the author used ChatGPT to assist with language editing, grammar improvement, formatting, and clarity of presentation. The author reviewed and edited the output as needed and take full responsibility for the content of the published article.

References

[1] M. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Comput. Surv.*, vol. 54, no. 8, pp. 1–41, 2022.

[2] T. Wu, Y. Xue, H. Li, S. Kumari, and L. Y. Por, "A survey of multi-blockchain: Architectures, technologies, and applications," *Comput. Mater. Contin.*, vol. 87, no. 3, 2026, doi: 10.32604/cmc.2026.077332.

[3] A. Fujihara, D. Morháč, and K. Koštál, "Understanding cross-chain message passing delays in Polkadot: An experimental perspective," in *Proc. 7th Int. Conf. Blockchain Computing and Applications (BCCA)*, 2025, pp. 225–232, doi: 10.1109/BCCA66705.2025.11229575.

[4] K. Koštál, D. Morháč, and J. Mečír, "Building interoperability: A decentralized bridge connecting Polkadot and Cosmos ecosystems," in *Proc. 37th Conf. Open Innovations Association (FRUCT)*, 2025, doi: 10.23919/FRUCT65909.2025.11008190.

[5] S. Chen, "Review on cross-chain bridges for data verification," in *Proc. 5th Int. Conf. Neural Networks, Information and Communication Engineering (NNICE)*, 2025, pp. 1558–1562, doi: 10.1109/NNICE64954.2025.11064377.

[6] R. B. S., S. K. Murthy, S. G. M., and S. S. P., "Cross-chain interoperability techniques and the role of zero-knowledge proofs," in *Proc. Int. Conf. Computing, Intelligence, and Application (CIACON)*, 2025, pp. 1–8, doi: 10.1109/CIACON65473.2025.11189597.

[7] K. Ding, X. Hu, W. Zhou, G. Cheng, R. Li, and H. Wu, "VeCroToken: An efficient, verifiable, and privacy-preserving cross-chain model for consortium blockchains based on zk-SNARKs," *IEEE Trans. Netw. Serv. Manag.*, vol. 23, pp. 1543–1558, 2026, doi: 10.1109/TNSM.2025.3636014.

[8] D. Mishra and S. Phansalkar, "Blockchain security in focus: A comprehensive investigation into threats, smart contract security, cross-chain bridges, and vulnerability detection tools and techniques," *IEEE Access*, vol. 13, pp. 60643–60671, 2025, doi: 10.1109/ACCESS.2025.3556499.

[9] S. Xu, D. Li, L. Wang, M. J. Mihaljević, S. Zhang, and W. Shao, "Cross-chain identity authentication protocol based on group signature and zero-knowledge proof," in *Proc. 28th Int. Conf. Computer Supported Cooperative Work in Design (CSCWD)*, 2025, pp. 2065–2070, doi: 10.1109/CSCWD64889.2025.11033396.

[10] O. Kuznetsov, Y. Khavikova, V. Bushkov, D. Shchytov, and N. Mormul, "Performance analysis of Groth16 zkSNARK: Systematic benchmarking with Circom-snarkjs," *Int. J. Comput.*, vol. 24, no. 4, pp. 645–660, 2025, doi: 10.47839/ijc.24.4.4329.

[11] J. Ernstberger et al., "zk-Bench: A toolset for comparative evaluation and performance benchmarking of SNARKs," *IACR Cryptology ePrint Archive*, Report 2023/1503, 2023. [Online]. Available: https://eprint.iacr.org/2023/1503

[12] Bright Inventions, "Zk-SNARKs in Substrate (Part 2): Using Groth16 and running Circom proof," Technical Blog, 2023. [Online]. Available: https://brightinventions.pl/blog/zk-snarks-in-substrate-part-2-using-groth16-and-running-circom-proof/

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[14] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," Ethereum White Paper, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[15] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," White Paper, 2016. [Online]. Available: https://polkadot.network/PolkaDotPaper.pdf

[16] J. Kwon and E. Buchman, "Cosmos: A network of distributed ledgers," White Paper, 2016. [Online]. Available: https://cosmos.network/whitepaper

[17] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from Bitcoin," in *Proc. IEEE Symposium on Security and Privacy*, 2014, pp. 459–474, doi: 10.1109/SP.2014.36.

[18] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. EUROCRYPT 2016*, Lecture Notes in Computer Science, vol. 9666, Springer, 2016, pp. 305–326, doi: 10.1007/978-3-662-49896-5_11.

[19] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *Proc. CRYPTO 2013*, Lecture Notes in Computer Science, vol. 8043, Springer, 2013, pp. 90–108, doi: 10.1007/978-3-642-40084-1_6.

[20] M. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-SNARK parameters in the random beacon model," *IACR Cryptology ePrint Archive*, Report 2017/1050, 2017. [Online]. Available: https://eprint.iacr.org/2017/1050

[21] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," *IACR Cryptology ePrint Archive*, Report 2019/953, 2019. [Online]. Available: https://eprint.iacr.org/2019/953

[22] J. Burdges, A. Cevallos, H. Kilinç Alper, C.-D. Liu-Zhang, F. Shirazi, A. Stewart, A. Gheorghe, R. Habermeier, R. Klotzner, B. Köcher, A. Ordian, M. K. Zyszkiewicz, and A. Sandu, "Efficient execution auditing for blockchains under Byzantine assumptions," *IACR Cryptology ePrint Archive*, Report 2024/961, 2024. [Online]. Available: https://eprint.iacr.org/2024/961

[23] M. Herlihy, "Atomic cross-chain swaps," in *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, 2018, pp. 245–254, doi: 10.1145/3212734.3212736.