ZK Move Rollup

Moved Network

moved.network

Abstract

The Move programming language offers a secure and auditable framework for developing smart contracts. However, existing Move-based blockchains face challenges with limited ecosystem traction and low liquidity bottlenecks. This whitepaper introduces Moved, a groundbreaking layer 2 zero-knowledge (ZK) rollup tailored for the Move language, enabling developers to build scalable decentralized applications while leveraging the liquidity and network effects of the Ethereum mainnet. Moved features a parallelized ZK proving architecture optimized to execute Move bytecode efficiently, achieving unprecedented scalability through advanced cryptography and execution models. By processing transactions off-chain in parallel and generating succinct ZK-Proofs, Moved inherits Ethereum's security while achieving orders-of-magnitude higher throughput than executing transactions directly on Ethereum. This paper presents Moved's architecture design, including the parallelized ZK infrastructure, optimized Move execution, state-of-the-art ZK-Proof system, and Merkelization techniques that minimize on-chain data footprint by keeping full account states off Ethereum Layer 1; only storing state and transaction hashes.

1 Introduction

The inception of Bitcoin in 2008 [8] ushered in a new era of decentralized and trustless computation facilitated by blockchain technology. While Bitcoin's core purpose was enabling peer-to-peer digital cash, it laid the foundational primitives for a more generalized blockchain framework. Ethereum [5],

introduced in 2015, built upon these primitives by incorporating a Turing-complete virtual machine, paving the way for executable smart contracts and decentralized applications (dapps). The Ethereum Virtual Machine (EVM) is a stack-based VM with a Turing complete instruction set that enables the deployment and execution of user-defined bytecode programs on the Ethereum blockchain. This revolutionary model of distributed computation rapidly catalyzed innovation, giving rise to decentralized finance (DeFi), non-fungible tokens (NFTs), decentralized autonomous organizations (DAOs), among other blockchain-native applications and use cases.

Solidity vs Move. While Solidity became the de facto language for Ethereum smart contract development, it suffers from inherent security risks stemming from semantic ambiguities, lack of formalism, and code complexity [12]. These vulnerabilities have led to numerous high-profile exploits resulting in substantial financial losses, most notably the infamous 2016 DAO attack [6] which drained around $60 million in ether at the time, as well as the more recent $190 million Nomad bridge hack [9] in 2022. The Move language [3], originally designed at Meta (Facebook) for the Diem blockchain, emerged as a safer and more robust alternative to Solidity. Move employs a bytecode interpretation execution model and follows a strict design philosophy focused on simplicity, auditability, and prevention of unintended behaviors. With influences from linear logic and secure coding principles, Move mitigates many of Solidity's pitfalls by preventing re-entrancy, data races, and other common vulnerabilities through its design that allows only a single execution context to access resources at any given time. Despite its security advantages, the Move ecosystem has faced challenges with limited liquidity, hindering widespread adoption compared to the more established Ethereum/Solidity landscape.

Moved Network Solution. To unlock Move's full potential while inheriting Ethereum's unparalleled liquidity and network effects, we introduce Moved - a novel zero-knowledge (ZK) rollup architecture tailored for the Move language. Moved enables Move smart contract execution by leveraging ZK virtual machine technology, allowing computations to be performed off-chain while allowing clients to know the execution is correct. Additionally, with the proofs and state merkle roots being posted to the Ethereum blockchain, clients of Moved benefit from the security of the Ethereum chain. Our initial ZK infrastructure builds upon Risc Zero's [4] general-purpose

zkVM, providing a secure and flexible foundation. Looking ahead, we plan to integrate Polygon's Miden VM [11], optimized specifically for efficient Move bytecode execution, unlocking even greater performance gains.

Central to Moved is a modular rollup design that prioritizes scalability, parallelization, and flexibility. We employ parallel execution techniques to maximize throughput, with the ability to run multiple Move virtual machines concurrently. Transaction data is aggregated into periodic ZK-Proofs that are settled on the Ethereum mainnet, inheriting its security guarantees. Our architecture remains flexible, enabling integration of the most efficient and available ZK tooling to provide fast finality. This agile approach ensures Moved can adapt to emerging innovations, consistently delivering a high-throughput layer 2 scaling solution as the ZK ecosystem evolves.

Outline. Rest of the paper is organized as follows. In Section 2 we give an overview of the Moved's modular rollup solution; in 3 we describe in detail how the ZK execution part of the system is designed; then lastly in Section 4 we describe the SDK for developer experience and detail the gas computation on the native ETH token.

2 Decentralized Rollup

The Moved rollup is composed of three modular components, each serving a specialized role as depicted in Figure 1. The Sequencer is responsible for ordering and batching incoming transactions, maintaining a consistent and tamper-proof transaction log. The provers run the batches of transactions generated by the sequencers to produce ZK proofs and state updates that are periodically submitted (e.g. every 12 hours) to the Verifier Contract. More details on the ZK execution the provers perform are given in the next section. The Verifier Contract, deployed on the Ethereum mainnet, acts as the ultimate arbiter, validating the ZK-Proofs against the corresponding Merkle roots and updating the canonical state on Layer 1. The Data Availability (DA) layer, such as Celestia [1], keeps the actual state that is behind the Merkle roots submitted to Layer 1 (Ethereum). Clients access the DA layer to query the state of the system (e.g. check what resources an account holds). The DA layer must provide a Merkle proof with its response so that the client can validate the state is correct using the Merkle root that is available from the Verifier Contract.

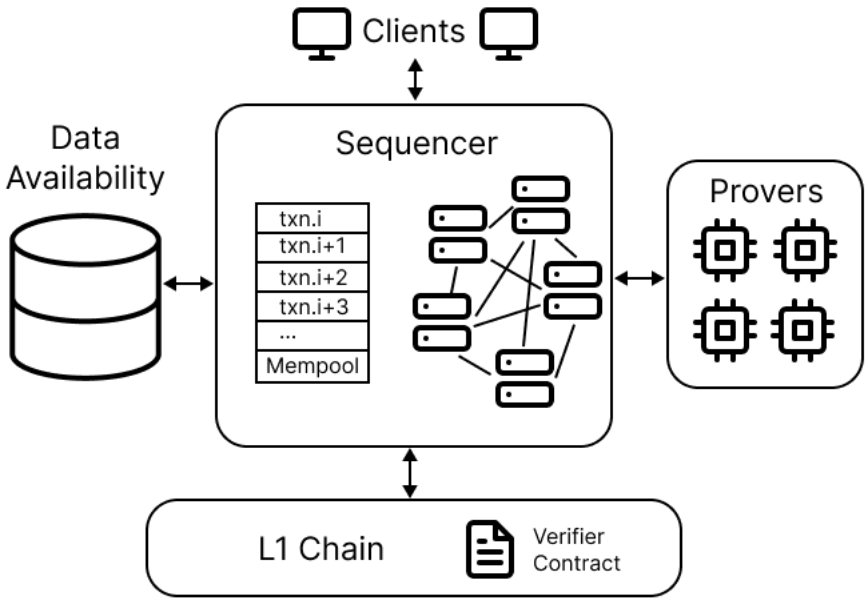


Figure 1: High-level architecture of the Moved ZK rollup, depicting the interaction between user applications, the sequencer, ZK virtual machines, data availability storage, and the Ethereum L1 verifier contract.

Modularity is a core tenet of the Moved rollup architecture, enabling continuous improvement and optimization of individual components. As more performant technologies emerge, we can seamlessly swap out and upgrade the Sequencer, Data Availability, or Verifier components without disrupting the overall system. Additionally, a dedicated Bridge Contract enables seamless liquidity flows between the Moved rollup and the Ethereum mainnet. Specifically, this bridge facilitates transfers of native ETH as well as ERC-20 standard tokens, allowing users to deposit assets from Ethereum into the Moved Network and withdraw them from Moved back to the Ethereum mainnet. In the initial phases, we plan to leverage specific, battle-tested rollup providers and infrastructure. Specifically, we intend to utilize Sovereign SDK [13] for their ZK Rollup support and integrate Celestia as the Data Availability layer. As the ecosystem matures, we will gradually introduce optimizations such as computing ZK-Proofs at the bytecode level to remove overhead and improve computational speed. Another optimization involves integrating a higher throughput sequencer to accelerate transaction ordering and improve overall system performance.

3 Zero Knowledge Execution

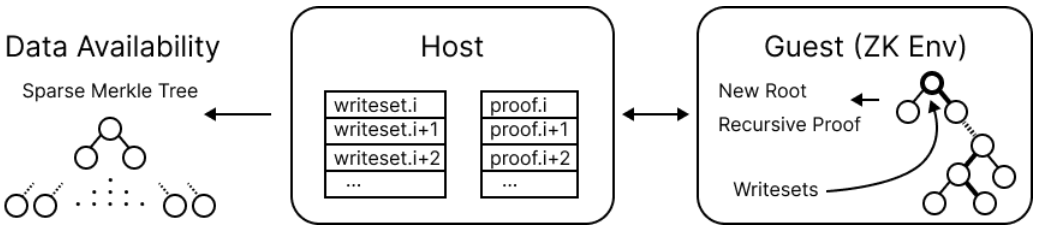
To achieve scalable Move smart contract execution while inheriting L1 security, Moved employs a ZK rollup architecture. ZK rollups leverage ZK-Proofs to validate the correctness of off-chain computations, enabling scalability by shifting the bulk of execution off the main Ethereum chain. Along with the ZK-Proof, the client obtains the Merkle root of the new state after execution of the batch of transactions. Both the proof and the new state root are sent the Ethereum network, effectively recording the validated execution of each Move transaction within the Moved rollup architecture while inheriting Ethereum's underlying security guarantees.

Our ZK execution will be developed in two phases. This allows us to get to market quickly while having an eye towards reducing our infrastructure costs in the future.

3.1 Phase 1

During this first phase, Moved utilizes a base Move virtual machine for smart contract execution, stripped from native code support found in chains like

(a) Step 1: Guest executions receive state and transaction data to run Move VM, producing state writeset and ZK-Proof outputs.



(b) Step 2: All the state writesets update Merkle tree and proofs are aggregated recursively.

Figure 2: Workflow of Risc Zero executing a Move smart contract to compute a new Merkle root representing the updated state.

Aptos [2] or Sui [7]. While direct ZK proving of Move bytecode execution will be added at a later stage, this approach allows us to establish a secure foundation for ZK-based Move execution quickly. The reason this approach allows for getting to market faster is because the Move VM is written in Rust which can be compiled and used in risc0 with minimal modification. This works because the Rust code can be compiled to the RISC-V instruction set architecture (ISA) and within the zkVM, the RISC-V code is executed in a ZK-friendly environment, generating cryptographic constraints that model the computational steps. Risc Zero's ZK-Proof system then takes these constraints and produces a succinct ZK-Proof, attesting to the correct execution of the Move smart contract transaction.

The Zero Knowledge execution process within Risc Zero involves the interaction between a non-ZK Host environment and a ZK-friendly Guest environment, as depicted in Figure 2. The Host retrieves all necessary state details from a Sparse Merkle Tree [10] representation and transmits them to the Guest. The goal of the ZK execution is to perform the state transition by

taking as input the current state root together with a batch of transactions, and producing as output the new state root after the transactions in that batch have been executed. The ZK execution proceeds in two steps.

First, all transactions (deployment of new Move contracts, or executions on existing ones) in the batch are executed in parallel on separate Guests. Within each Guest, the Move smart contract is executed by running the Move virtual machine inside a ZK computation model. The output from each of these executions is a changeset containing the state changes to be applied across different account resources. If there are read/write conflicts caused by the parallel execution then some transactions will need to be re-executed with a state that includes the changes from earlier transactions in the batch. Once the batch finishes processing we will have a proof of the execution for each transaction in the batch.

Then in the second step of the ZK execution we use risc0's recursive ZK-proving ability to include all those individual proofs into a single proof for the whole batch. This combined proof also includes applying all the changes to the state from the individual changes sets and computing the updated Merkle root. This updated root, along with the ZK proof that is was correctly produced, is the output of the ZK execution.

3.2 Phase 2

While the initial Moved architecture relies on Risc Zero's general zkVM, we plan to implement Move-optimized ZK execution in a future phase.

Move Bytecode Level ZK Execution. This involves building a Moved compiler that generates ZK circuit representations (Miden code) tailored for specific Move bytecode operations. The Miden code will be executed within a ZK virtual machine, generating granular ZK-Proofs at the individual bytecode instruction level. This bytecode-level ZK execution model will greatly reduce the computational resources needed to run the provers because we will no longer have the overhead of the whole Move interpreter. Critically, these Phase 2 architectural enhancements will be designed for backward compatibility, allowing a seamless transition while maintaining support for the initial ZK execution pipeline.

Achieving bytecode-level ZK execution requires a specialized compiler toolchain that can analyze and translate individual Move bytecode instructions into an optimized ZK assembly representation. Moved's compiler parses

```
Bytecode::Add => Node::Instruction(Instruction::Add),
Bytecode::Sub => Node::Instruction(Instruction::Sub),
Bytecode::Mul => Node::Instruction(Instruction::Mul),
Bytecode::Div => Node::Instruction(Instruction::U32Div),
```

Figure 3: Illustration of a straightforward mapping from Move bytecode instructions to the equivalent Miden ZK assembly representations.

through each Move bytecode operation, methodically converting it into equivalent ZK assembly code tailored for the Miden virtual machine. This conversion process involves a mix of direct bytecode mappings for straightforward instructions (as illustrated in Figure 3) as well as complex mappings for those that require more intricate ZK constraint modeling (as the conversion steps shown in Figure 4).

Once the entire Move bytecode has been transformed into the ZK assembly representation, the resulting code essentially becomes the ZK smart contract deployed within the Miden execution environment. When users initiate transactions, the corresponding ZK assembly instructions are loaded and executed within the ZK virtual machine's constrained CPU. This ZK execution model generates succinct proofs at the granular bytecode operation level, attesting to the correct computational steps. Additionally, Moved's compiler performs further optimizations on the final ZK assembly, enhancing execution efficiency by minimizing redundant constraints and leveraging batch processing where applicable.

4 Moved SDK and Gas

Moved Network will provide an SDK in multiple programming languages to facilitate developers interacting with our system. In our system architecture, we prioritize seamless integration with widely adopted Ethereum wallets, facilitating smooth transaction signing and transmission to the Sequencer. To achieve this, we ensure compatibility with Ethereum RPC endpoints, enabling robust support for these wallets. It will also query the data availability layer to get details on accounts. All of this is done by connecting to the Moved Network's Ethereum-standard RPC endpoints.

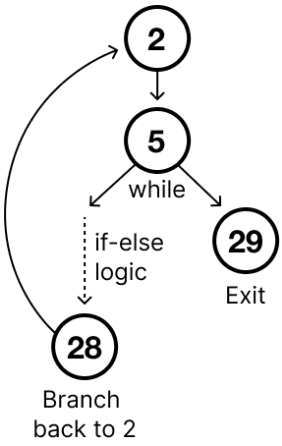
An important aspect of the Moved SDK is accurately computing the gas costs associated with executing transactions on the rollup network. There

```
fun collatz(n: u32): u32 {
  let count: u32 = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    };
    count = count + 1;
  };
  count
}
```

(a) Collatz conjecture sequence calculation in Move

```
vec![
  Bytecode::LdU32(0),
  Bytecode::StLoc(1),
  Bytecode::CopyLoc(0),   (2)
  Bytecode::LdU32(1),
  Bytecode::Neq,
  Bytecode::BrFalse(29), (5)
  ...
  Bytecode::Branch(2),   (28)
  Bytecode::MoveLoc(1), (29)
  Bytecode::Ret,
]
```

(b) Generated Move bytecodes with line numbers in parantheses



(c) Corresponding Control Flow Graph between lines

```
proc.collatz
  ...
  while.true
    // if-else logic
    ...
  end
  // Ln 29: Save in memory, exit
  mem_store.COLLATZ_INDEX
end
```

(d) Generated Miden assembly displaying only the branching sections

Figure 4: Move source code (a) is compiled to bytecode with branches (b), which constructs a Control Flow Graph (c) representing state transitions, guiding the heuristic generation of optimized Miden assembly (d).

are two primary fee components - the cost of data availability storage and the cost of settling state proofs on Ethereum Layer 1. Gas fees are denominated and paid in ETH tokens.

When initiating a transaction, developers can leverage the *estimateGas* API which provides an estimate of the gas contribution from their transaction to the overall batch size. The base fee is the minimum price per transaction, primarily accounting for storage costs on Layer 1 and the data availability layer. Separately, the execution gas is a dynamic fee calculated by the Moved compiler based on the complexity of the Move bytecode execution. This portion is paid to the Sequencer and Provers to compensate for the computational resources consumed in processing transactions and maintaining operational overheads. Through this multi-component gas model, Moved ensures an equitable distribution of fees across different layers of the rollup architecture.

5 Conclusion

The Moved rollup represents a groundbreaking integration of the Move programming language with Ethereum's liquidity and network effects through a novel zero-knowledge architecture. By enabling secure off-chain computation of Move smart contracts while ensuring validity on Ethereum, Moved overcomes scalability limitations plaguing existing Move blockchains. With a robust ZK virtual machine core optimized for efficient Move bytecode execution, and a decentralized rollup design prioritizing modularity, Moved establishes a paradigm shift for high-throughput, trustless application development. As adoption grows, Moved will cultivate an interconnected ecosystem allowing developers to seamlessly access Ethereum's mature liquidity sources while building on Move's security principles, driving blockchain scalability into uncharted frontiers.

References

[1] M. Al-Bassam. Lazyledger: A distributed data availability ledger with client-side smart contracts, 2019.

[2] Aptos Labs. The aptos blockchain: Safe, scalable, and upgrade-

able web3 infrastructure, 2022. URL https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf.

[3] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou. Move: A language with programmable resources, 2020. URL https://developers.diem.com/docs/technical-papers/move-paper. Accessed on 2024-03-20.

[4] J. Bruestle, P. Gafni, and the RISC Zero Team. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity, Aug 2023. URL https://dev.risczero.com/proof-system-in-detail.pdf.

[5] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL https://github.com/ethereum/wiki/wiki/White-Paper.

[6] P. Daian. Analysis of the dao exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, 2016. Accessed: 2024-03-20.

[7] Mysten Labs. The sui smart contracts platform, 2022. URL https://docs.sui.io/paper/sui.pdf.

[8] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. URL https://bitcoin.org/bitcoin.pdf.

[9] Nomad. Nomad bridge incident. https://nomad.xyz/incident.html, 2022. Accessed: 2024-03-20.

[10] R. Östersjö. Sparse merkle trees: Definitions and space-time trade-offs with applications for balloon. 2016. URL https://www.diva-portal.org/smash/get/diva2:936353/FULLTEXT01.pdf.

[11] Polygon Miden. Stark-based virtual machine. https://github.com/0xPolygonMiden/miden-vm, 2024.

[12] Solidity. Security considerations, 2016. URL https://docs.soliditylang.org/en/latest/security-considerations.html. Accessed on 2024-03-20.

[13] Sovereign SDK. The internet of rollups. `https://sovereign.xyz`. Accessed: 2024-03-20.