# ZK Input Passthrough: Secure Input Verification for Off-Chain Computations in Blockchain Systems

Arthur Guiot
Director of Engineering
aguiot@scu.edu

Zach Teal
Director of Research
zteal@scu.edu

April 16, 2025

**Abstract**

Zero-Knowledge Virtual Machines (ZK VMs) enable verifiable off-chain computations in blockchain ecosystems, but ensuring input authenticity remains a significant challenge. This paper introduces the "ZK Input Passthrough" method, which combines on-chain hash verification with ZK coprocessors to guarantee input integrity without revealing the underlying data. Our approach, implemented using the SP1 stack[2], provides a practical solution for decentralized applications requiring trustworthy off-chain computations while maintaining data privacy. We demonstrate the effectiveness of our method through a reference implementation and analyze its security properties in the context of blockchain applications.

## 1 Introduction

The blockchain ecosystem increasingly relies on zero-knowledge proofs (ZKPs) for scalability and privacy. As blockchain applications grow in complexity, the need for efficient off-chain computation becomes paramount. Zero-Knowledge Virtual Machines (ZK VMs), particularly SP1[2] and RISC0[4], have emerged as powerful tools for verifiable off-chain computation. These systems allow complex calculations to be performed outside the blockchain while maintaining cryptographic guarantees about the correctness of the computation.

However, a critical challenge remains unaddressed: while ZK VMs can prove that a computation was executed correctly, they cannot inherently verify the authenticity of the input data. This limitation creates a significant vulnerability where a malicious actor could provide false inputs, resulting in proofs that are technically valid but based on fraudulent data. The challenge of input verification in zero-knowledge systems has been studied extensively[1], but existing solutions often require complex trusted setups or rely on centralized authorities.

Our approach builds upon the fundamental principles of hash chains[3] while introducing novel mechanisms for input verification in the context of ZK VMs. To illustrate the practical importance of this problem, consider a decentralized finance application computing market volatility metrics using cryptocurrency price data. While the computation must occur off-chain due to gas constraints, the system must guarantee that the input prices are genuine. Traditional approaches might rely on trusted oracles or complex multi-party computation schemes, each introducing their own set of challenges and trust assumptions.

The ZK Input Passthrough method addresses these challenges by creating a verifiable link between on-chain recorded data and off-chain computations. Our solution leverages the immutability and transparency of blockchain systems while maintaining the efficiency benefits of off-chain computation. By combining hash-based verification with ZK proofs, we achieve a robust system that ensures input integrity without compromising on performance or security.

## 2 ZK Input Passthrough Methodology

Our method comprises three interconnected components that ensure input integrity while maintaining compatibility with existing ZK VM frameworks. The system leverages the security properties of cryptographic hash functions[1] and the verification capabilities of modern ZK VMs[2]. This section details the core components and their interactions within the broader system architecture.

### 2.1 Data Generation and Hash Storage

The foundation of our approach lies in the secure generation and storage of data hashes on-chain. This process must accommodate both sequential and non-sequential data patterns, each requiring different handling mechanisms to maintain security and efficiency. The system implements two distinct but complementary approaches to handle these different data types.

#### 2.1.1 Sequential Data Processing

For time-series data, such as oracle price feeds or regular market updates, we implement a linear hash chain inspired by Merkle's work[3]. This approach is particularly effective for applications requiring temporal consistency and ordered data verification. The hash chain is constructed as:

$$H_n = \text{keccak256}(D_n \parallel H_{n-1})$$

where $H_n$ represents the current hash, $D_n$ represents the current data point, $H_{n-1}$ is the previous hash, and $\parallel$ denotes concatenation. This construction ensures that each new hash incorporates both the current data and the entire history of previous inputs through the previous hash value.

The sequential nature of this approach provides several advantages. First, it creates an unbroken chain of verification that makes it impossible to modify historical data without detection. Second, it allows for efficient verification of data ordering, which is crucial for applications requiring temporal consistency. Third, it minimizes storage requirements as only the latest hash needs to be maintained for verification purposes.

#### 2.1.2 Non-Sequential Data Handling

For standalone data points or non-sequential information, such as user credentials or discrete events, we employ a simpler but equally secure approach:

$$H = \text{keccak256}(D)$$

This direct hashing approach is suitable for cases where temporal ordering is not critical, or when data points are independent of each other. The system maintains these hashes alongside relevant metadata such as timestamps and usage flags, enabling verification without requiring the maintenance of a hash chain.

The hash storage contract serves as the on-chain anchor for our system, maintaining these values with associated timestamps and metadata. This contract leverages Ethereum's native keccak256 implementation[1], ensuring consistent hash computation across both on-chain and off-chain components of the system.

### 2.2 Off-Chain ZK Computation with Input Passthrough

The ZK coprocessor circuit forms the core of our off-chain computation system, performing three essential operations that work in concert to ensure both computational correctness and input validity. This component must balance the competing requirements of computational efficiency and security guarantees.

The first operation involves hash verification, where the circuit reconstructs and validates either the hash chain or standalone hash using the same cryptographic primitives as Ethereum[1]. This step ensures that the input data matches what was recorded on-chain, preventing manipulation of inputs during the off-chain computation phase.

Following verification, the circuit executes the intended calculation within the ZK VM environment[2]. This computation can range from simple arithmetic operations to complex financial calculations, all while maintaining zero-knowledge properties. The ZK VM ensures that the computation itself is verifiable without revealing the specific inputs or intermediate values.

Finally, the circuit produces two crucial outputs: the computation result and the verified hash. This dual output approach allows the on-chain contract to verify both the correctness of the computation and the authenticity of the inputs used, creating a complete verification chain.

### 2.3 On-Chain Verification

The final component of our system involves on-chain verification, which provides the ultimate security guarantee for our input passthrough method. This process combines two distinct but complementary verification steps that together ensure the integrity of both the computation and its inputs.

The first step involves ZK proof validation using SP1's native verification contracts[2]. This verification ensures that the computation was performed correctly according to the specified circuit logic, leveraging the security properties of the underlying zero-knowledge proof system.

The second step involves hash comparison, where the passthrough hash included in the circuit output is matched against the on-chain records maintained by the data collection contract. This step closes the verification loop, ensuring that the inputs used in the off-chain computation exactly match those recorded on-chain.

## 3 Implementation

Our reference implementation demonstrates the practical viability of the ZK Input Passthrough method using the SP1 stack[2]. This section details the key components of our implementation, discussing design decisions and technical considerations that arise when deploying the system in a production environment.

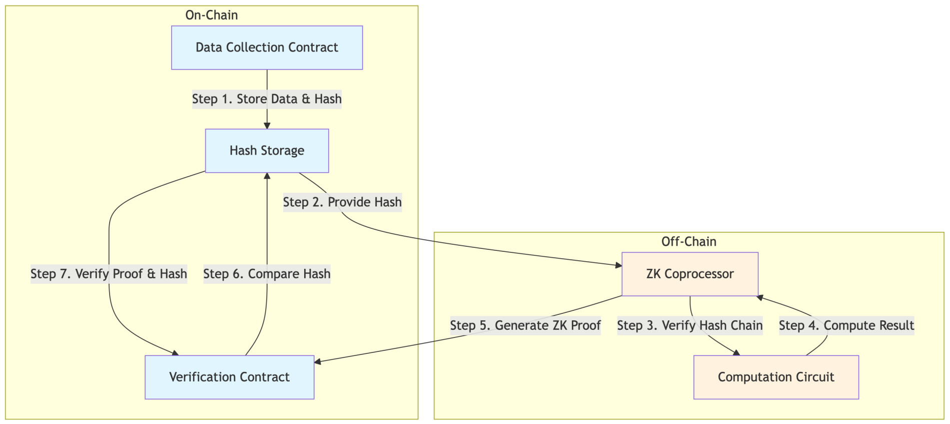

Figure 1: Architecture of ZK Input Passthrough showing data flow and verification steps

## 3.1 Data Collection Contract

The data collection contract serves as the primary on-chain component of our system, responsible for maintaining the integrity of input data through hash storage and verification. Our implementation uses Solidity's native types and operations to ensure efficient gas usage while maintaining robust security properties.

The contract implements a sophisticated data structure that tracks not only the hashes but also their usage and temporal relationships:

Listing 1: Data Collection Smart Contract

```solidity
contract DataCollection {
    struct DataEntry {
        bytes32 data;
        bytes32 hash;
        uint256 timestamp;
        bool used;
    }

    mapping(uint256 => DataEntry) public entries;
    uint256 public currentIndex;
    bytes32 public latestHash;

    event NewDataEntry(uint256 indexed index, bytes32 hash);

    function updateData(bytes32 data) public {
        bytes32 newHash = keccak256(
            abi.encodePacked(data, latestHash)
        );

        entries[currentIndex] = DataEntry({
            data: data,
            hash: newHash,
            timestamp: block.timestamp,
            used: false
        });

        latestHash = newHash;
        emit NewDataEntry(currentIndex++, newHash);
    }

    function verifyHash(bytes32 hash) public view returns (bool) {
        return hash == latestHash;
    }
}
```

The contract's design incorporates several important features. The DataEntry struct captures not only the hash and data but also includes a timestamp and usage flag, enabling temporal verification and preventing replay attacks. The mapping structure provides efficient access to historical entries while maintaining a clear sequence through the currentIndex counter.

The updateData function implements the hash chain logic, combining new data with the previous hash to maintain the chain's integrity. This function also emits events, allowing external systems to track updates and maintain synchronization with the contract state. The verifyHash function provides a simple but crucial interface for the on-chain verification of passthrough hashes.

## 3.2 ZK Circuit Implementation

The zero-knowledge circuit implementation represents the core of our off-chain computation system. Written in Rust for the SP1 platform, the circuit combines hash verification with the primary computation logic:

Listing 2: SP1 Circuit Implementation

```rust
#[circuit]
pub struct InputVerificationCircuit {
    data: Vec<u8>,
    previous_hash: [u8; 32],
    claimed_hash: [u8; 32],
}

impl Circuit for InputVerificationCircuit {
    fn verify(&self) -> bool {
        // Compute hash chain
        let computed = self.compute_hash_chain();

        // Verify hash matches
        computed == self.claimed_hash
            && self.verify_computation()
    }

    fn compute_hash_chain(&self) -> [u8; 32] {
        keccak256(&[
            &self.data[..],
            &self.previous_hash[..]
        ].concat())
    }
}
```

The circuit implementation demonstrates several key design principles. First, it separates the hash verification logic from the main computation, allowing for modular development and easier maintenance. The verify function combines both hash verification and computation verification in a single boolean expression, ensuring that both conditions must be satisfied for the circuit to be valid.

The compute_hash_chain function implements the same hashing logic as the on-chain contract, ensuring consistency between on-chain and off-chain hash computation. This implementation detail is crucial for maintaining the security of the input passthrough mechanism.

## 3.3 Integration Considerations

The integration of these components requires careful attention to several technical details. First, the system must handle gas optimization for on-chain operations, particularly in the data collection contract. Our implementation uses efficient data structures and minimal storage operations to reduce gas costs.

Second, the ZK circuit must be designed to handle various edge cases, such as initial states where no previous hash exists, or scenarios where data verification might fail. Our implementation includes robust error handling and state validation to ensure reliable operation under all conditions.

Third, the system must maintain synchronization between on-chain and off-chain components. This is achieved through careful event handling and state management, ensuring that the ZK circuit always has access to the correct historical data and hash values.

## 4 Security Analysis

The security of ZK Input Passthrough builds upon established cryptographic principles while introducing novel considerations specific to blockchain environments. This section provides a comprehensive analysis of the security properties, potential vulnerabilities, and their mitigations within our system.

### 4.1 Hash Chain Properties

The fundamental security of our system relies on the cryptographic properties of the hash chain implementation. The use of keccak256[1] as our hash function provides three critical security properties that form the foundation of our input verification mechanism.

Pre-image resistance ensures that it is computationally infeasible to derive the original input data from its hash value. This property is crucial for maintaining data privacy in our system, as only hashes are stored on-chain while the actual data remains confidential. Even if an attacker gains access to the complete hash chain, they cannot reconstruct the original input data without access to the specific pre-images.

Collision resistance prevents attackers from finding alternative inputs that produce the same hash value. This property is essential for preventing fraud, as it ensures that a malicious actor cannot construct alternative data that would pass our verification checks. The computational difficulty of finding such collisions in keccak256 provides strong security guarantees for our system.

The linking property of our hash chain construction ensures that each hash depends on all previous data points in the sequence. This temporal consistency is crucial for applications requiring ordered data verification, such as price feed systems or sequential transaction processing. Any attempt to modify historical data would break the hash chain, making such tampering immediately detectable.

### 4.2 Attack Vectors and Mitigations

Our security analysis identifies several potential attack vectors and implements specific mitigations for each. Understanding these attack vectors is crucial for both system implementers and users.

Replay attacks represent a significant threat to any input verification system. An attacker might attempt to reuse valid historical inputs to manipulate the system's state. Our implementation addresses this through a combination of techniques. The data collection contract maintains explicit usage flags for each hash entry, preventing multiple uses of the same input. Additionally, the temporal metadata associated with each entry allows the system to enforce time-based constraints on input validity.

Race conditions present another challenge, particularly in blockchain environments where transaction ordering is not guaranteed. An attacker might attempt to exploit the gap between hash submission and verification to manipulate the system state. Our implementation handles this through careful transaction ordering enforcement and atomic operations where possible. The timestamp management system ensures that inputs are processed in the correct sequence, while the blockchain's consensus mechanism provides additional protection against such attacks.

Hash chain manipulation attempts are prevented through our comprehensive verification process. The on-chain verification step ensures that only hashes recorded in the data collection contract can be used as inputs to the ZK computation. Any attempt to manipulate the hash chain would require modifying the blockchain's state, which is prevented by the underlying consensus mechanism.

Front-running attacks, where an attacker observes pending transactions and attempts to insert their own transactions ahead of them, are mitigated through several mechanisms. Our system implements commitment schemes where appropriate, and the atomic nature of our verification process ensures that front-running attempts cannot compromise input integrity.

### 4.3 Trust Model and Assumptions

Our security model makes several important assumptions that should be understood by implementers. While we don't assume the reliability of individual data sources, we do assume that the selected data provider (such as an oracle for price data) is operating correctly. This assumption is reasonable in practice, as the choice of data provider is typically part of the system's governance process.

The system also relies on the security properties of the underlying blockchain platform and the correctness of the ZK VM implementation. While these are standard assumptions in blockchain applications, they should be explicitly considered when evaluating the system's security.

### 4.4 Formal Security Properties

The security of our system can be formally characterized through several key properties:

1. **Input Integrity**: No attacker can forge valid inputs that weren't previously recorded on-chain.

2. **Temporal Consistency**: The ordering of inputs in the hash chain cannot be modified without detection.

3. **Non-Repudiation**: Once an input is used in a computation, this usage cannot be denied or modified.

4. **Privacy**: The system reveals no information about the input data beyond what is explicitly shared through the computation output.

These properties together ensure that our system provides robust security guarantees while maintaining the practical utility necessary for real-world applications.

## 5 Future Work

The development of ZK Input Passthrough opens several promising avenues for future research and development. These directions aim to enhance the system's capabilities, improve its performance, and broaden its applicability across different blockchain applications.

The integration of Merkle tree structures represents a significant opportunity for scaling our system to handle larger datasets. While our current implementation uses a linear hash chain, Merkle trees[3] could provide more efficient proof generation and verification for non-sequential data. This approach would be particularly valuable for applications dealing with large-scale data verification, such as decentralized identity systems or complex financial instruments.

Performance optimization remains an important area for future work. The current implementation adds some computational overhead through hash verification in ZK circuits. Research into more efficient hash function implementations specifically designed for ZK circuits could significantly reduce this overhead. Additionally, exploring batch processing techniques could improve throughput for applications requiring multiple input verifications.

The standardization of interfaces for ZK VM integration presents another crucial direction. As the ecosystem of ZK VMs continues to grow, developing common standards for input verification would facilitate interoperability and reduce implementation complexity. This standardization effort could include common interfaces for hash verification, data submission, and proof generation across different ZK VM platforms.

Advanced privacy features could further enhance the system's utility. While our current implementation maintains basic input privacy, additional techniques from the zero-knowledge literature could be incorporated to provide stronger privacy guarantees. This might include selective disclosure mechanisms or advanced commitment schemes that allow for more flexible privacy policies.

Research into dynamic data sources and verification mechanisms could expand the system's applicability. Current implementations assume relatively static data sources, but many applications require dynamic source selection or multi-source verification. Developing mechanisms to securely handle these scenarios while maintaining our security guarantees presents an interesting challenge.

## 6 Conclusion

ZK Input Passthrough represents a significant advancement in securing off-chain computations for blockchain applications. By combining hash-based verification with zero-knowledge proofs, our method provides robust input integrity guarantees while maintaining the efficiency benefits of off-chain computation.

The implementation demonstrates the practical viability of our approach, showing that it can be effectively deployed using existing ZK VM technology. Our security analysis provides confidence in the system's robustness against various attack vectors, while identifying clear boundaries for its security guarantees.

The system's ability to handle both sequential and non-sequential data makes it applicable to a wide range of use cases, from financial applications requiring market data verification to identity systems needing secure credential validation. The flexibility of our approach allows it to be adapted to different requirements while maintaining its core security properties.

As blockchain applications continue to evolve and require more complex off-chain computations, the importance of secure input verification will only grow. ZK Input Passthrough provides a foundation for addressing these challenges, offering a practical solution that can be built upon and extended to meet future needs.

## Acknowledgments

This research was supported by Santa Clara University's Blockchain Initiative. We acknowledge the innovative work of the SP1 technology, which made this implementation possible.

## References

[1] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference. *Submission to NIST (Round 3)*, 510(1), 2011.

[2] Succinct Labs. Sp1: A performant, 100% open-source, contributor-friendly zkvm. `https://github.com/succinctlabs/sp1`, 2023. Accessed: 2024-02-23.

[3] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer, 1987.

[4] RISC Zero. Risc zero: A zero-knowledge virtual machine. `https://www.risczero.com/`, 2023. Accessed: 2024-02-23.