Implementation of a Custom Hash Function and Blockchain Deployment on XDC Apothem Network

# Abstract

This is a report on a project titled 'Multiple User Authentication with MetaMask Verification,' a blockchain decentralized application (DApp) that demonstrates the use of Ethereum smart contracts and MetaMask for secure verification. The general goal of the project is to create a trustless verification system whereby multiple users can verify themselves without the use of centralized servers. The process begins with the creation of the hash function and continues with the creation of smart contracts in Remix, security attack testing, deployment in a local Hardhat blockchain on Ubuntu, migration to the XDC Apothem testnet, and final verification with the inclusion of MetaMask integration This report contains the motivation, design, implementation, results, and future improvements for the project.

# Introduction

Blockchain has been one of the most promising areas of computer science and information technology innovation in the past few years. Decentralized applications (DApps) are conquering the globe quickly because they eliminate intermediaries and bring transparency, security, and trust. Authentication is an essential feature of any application. The conventional methods of authentication employ usernames, passwords, or centralized identity providers, which are all breakable and hackable. To overcome these challenges, this project suggests a blockchain-based authentication system built around MetaMask, a popular Ethereum wallet and gateway to blockchain applications.

# Objectives

* **To develop and implement a customized hash function**
* Design a self-made hash function from scratch without relying on pre-built cryptographic libraries.
* Incorporate **bitwise operations (AND, OR, XOR, NOT), rotation, and modular arithmetic** to mimic real cryptographic hash behaviors.
* Apply padding techniques to ensure that input messages are aligned to block sizes.
* This objective builds foundational knowledge of cryptography and helps in understanding how popular algorithms like SHA or MD5 are structured internally.
* **To develop and deploy a Solidity smart contract in Remix**
* Write Solidity smart contracts that incorporate the custom hash function for verification.
* Test and debug the contracts within **Remix IDE**, which provides a user-friendly environment before moving to more complex local setups.
* Ensure contract functionality such as secure storage, verification, and interaction logic is working correctly at this stage.
* **To simulate security attacks and test contract resiliency**

Identify and simulate common blockchain vulnerabilities, such as:

* **Reentrancy attacks** (calling back into the contract before execution completes).
* **Denial of Service (DoS)** attacks through malicious inputs.
* **Unauthorized access** attempts to modify or retrieve restricted data.

Analyze the behavior of the system under such conditions and strengthen the smart contract with protective coding practices (e.g., checks-effects-interactions pattern, modifiers, require statements).

* **To run the system locally with Hardhat on Ubuntu**
* Use **Hardhat**, a popular Ethereum development framework, for local blockchain simulation.
* Deploy contracts in a controlled environment, allowing safe debugging and performance testing.
* Run unit tests, script-based deployment, and verify contract behavior locally before pushing to a live testnet.
* **To include deployment to the XDC Apothem testnet with MetaMask support**
* Move from local simulation to a public test environment by deploying on the **XDC Apothem Testnet**.
* Configure **MetaMask** to connect to the Apothem network and enable real transaction signing.
* Provide a near real-world testing scenario where users interact with the DApp as they would on a production blockchain.
* **To include the custom hash function for secure verification on-chain**
* Integrate the previously built custom hash function directly into the Solidity smart contracts.
* Use the hash for **message verification, transaction authentication, and integrity checks** within the blockchain system.
* This ensures that the DApp not only relies on blockchain immutability but also adds an extra layer of cryptographic security
* **To develop a frontend interface for process visualization**
* Build a **React-based frontend application** that interacts with the deployed smart contracts.
* Allow users to connect via MetaMask and visualize the entire workflow, including message hashing, contract verification, and transaction flow.
* Provide an intuitive interface so that even non-technical users can see how blockchain verification and authentication processes are carried out.

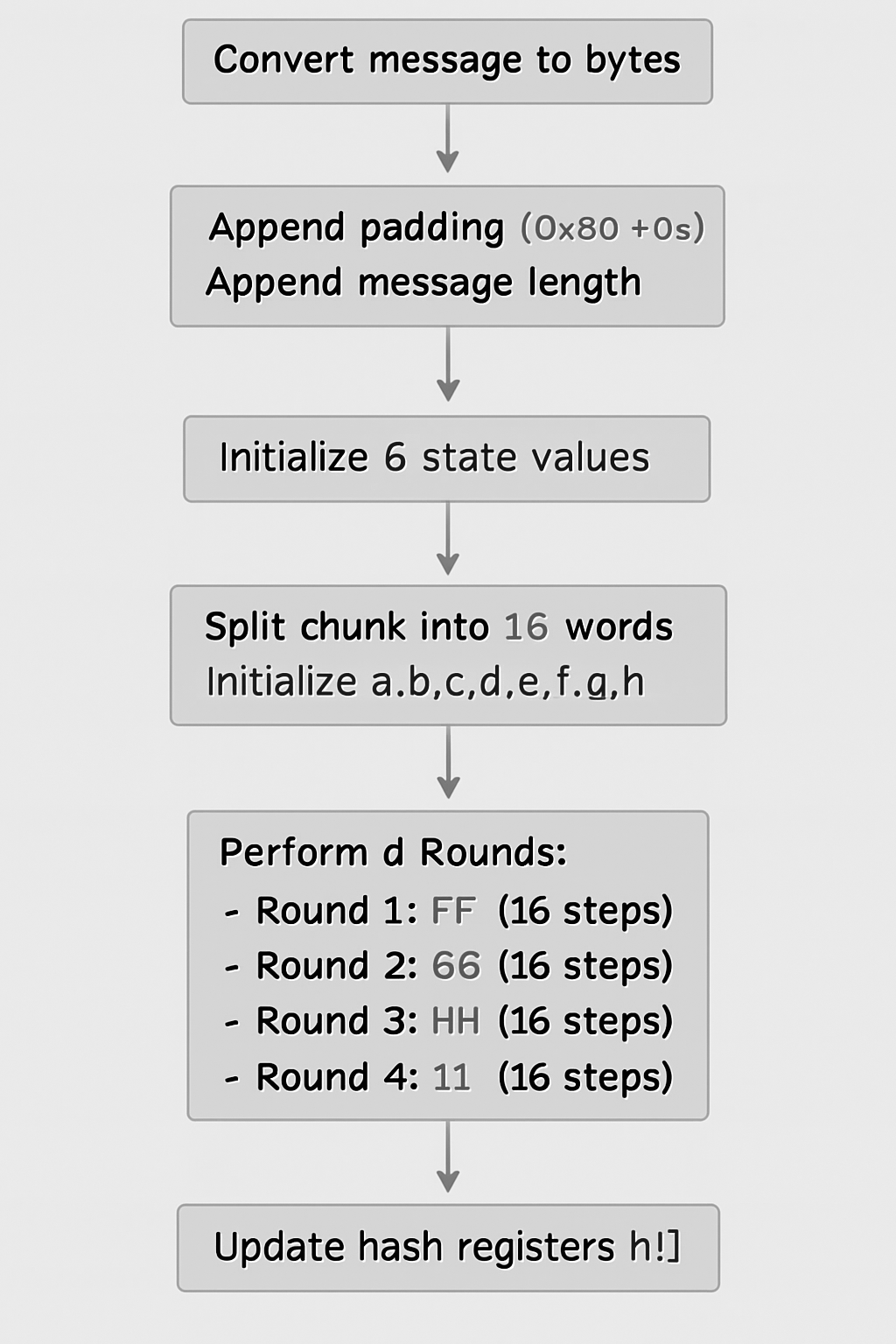
## Methodology

### Custom Hash Function

### The first step in the project involved the design and implementation of a **custom cryptographic hash function**. The function was initially developed in **Python** to test its correctness, performance, and memory usage. Later, it was adapted into **JavaScript** for integration with the blockchain-based smart contract.

The algorithm works on a **block size of 64 bytes**, similar to popular cryptographic functions such as MD5 and SHA. The process follows a systematic flow:

1. **Padding** – The input message is padded to ensure that its length is compatible with the block size.
2. **Splitting** – The message is divided into fixed-size blocks for processing.
3. **Logical Operations** – Each block undergoes operations such as XOR, bitwise shifts, and circular rotations to increase diffusion and randomness.
4. **Compression** – Intermediate results are compressed into smaller fixed-length outputs.
5. **Digest Generation** – Finally, a **fixed-size output** (e.g., 128-bit/256-bit) is produced, representing the unique hash of the input.

This function demonstrates the working principles of cryptographic primitives and forms the basis for **secure on-chain verification**.

### Smart Contract Development (Remix IDE)

The second step involved the creation of a smart contract in Solidity using the Remix IDE. The initial development involved creating a simple contract that could carry out simple storage and retrieval operations. The contract was later adjusted to accept hash values as input from users, thereby linking it to the custom hash function.

Testing was performed using multiple accounts present in the Remix environment such that multiple users could submit and authenticate their respective hash values. The contract logic was verified to be correct in this phase prior to going ahead towards local deployment.

***Smart Contract Code:***

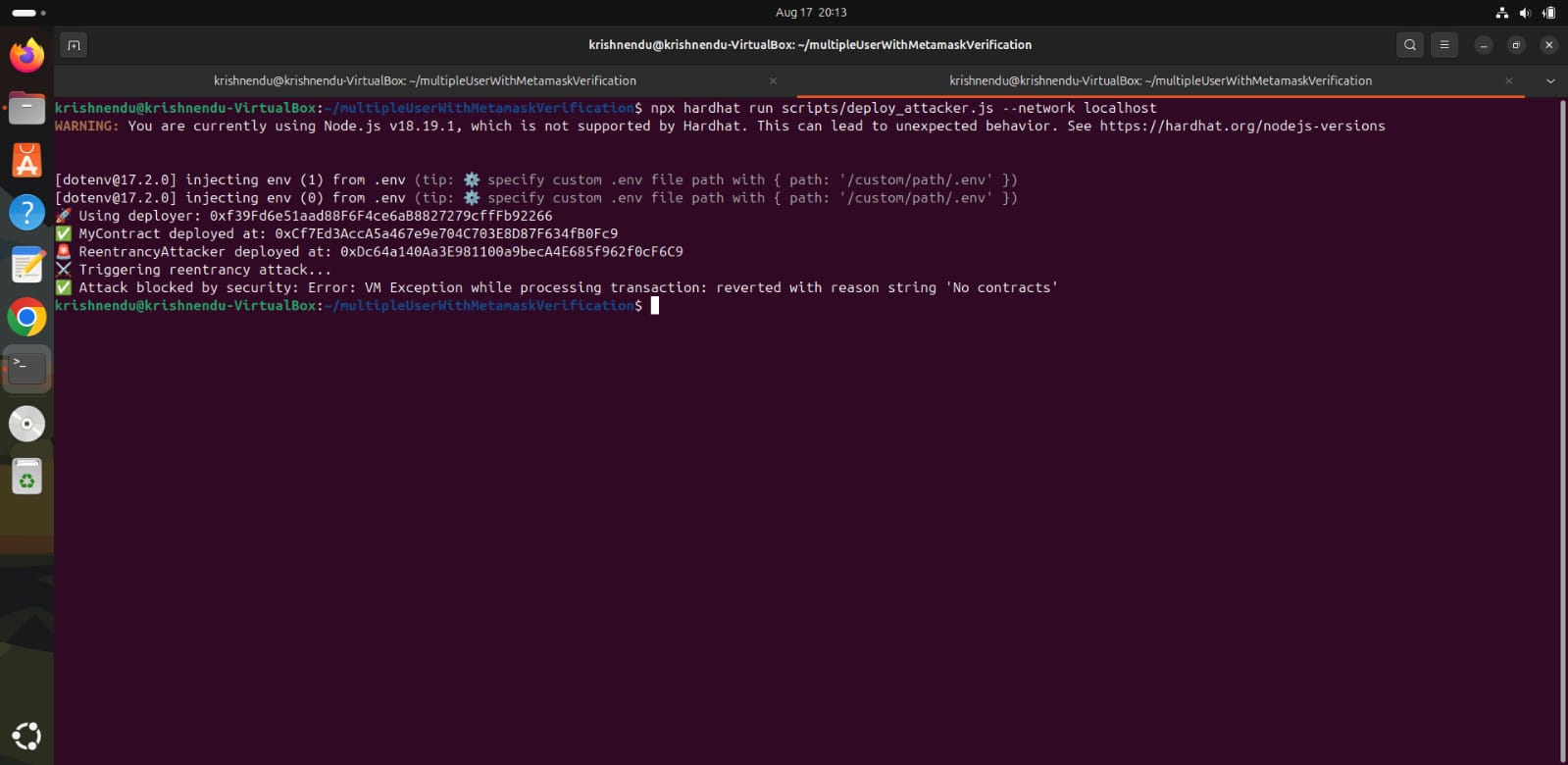

### Security Testing

Smart contracts deployed on blockchain networks are **irreversible** in nature; once deployed, any vulnerabilities may be exploited permanently. Hence, security testing was an essential part of the methodology.

The following **attacks were simulated**:

* **Reentrancy Attack** –

A reentrancy attack occurs when an external contract is able to repeatedly call back into a vulnerable contract before the first invocation is fully resolved. This can lead to draining funds or bypassing state update logic. In the project, a reentrancy attempt was simulated using fallback calls to re-invoke setMessage. Although the contract had no financial transfer, the test demonstrated how recursive calls could, in principle, overwrite stored values or bypass restrictions if not handled with checks-effects-interactions design.



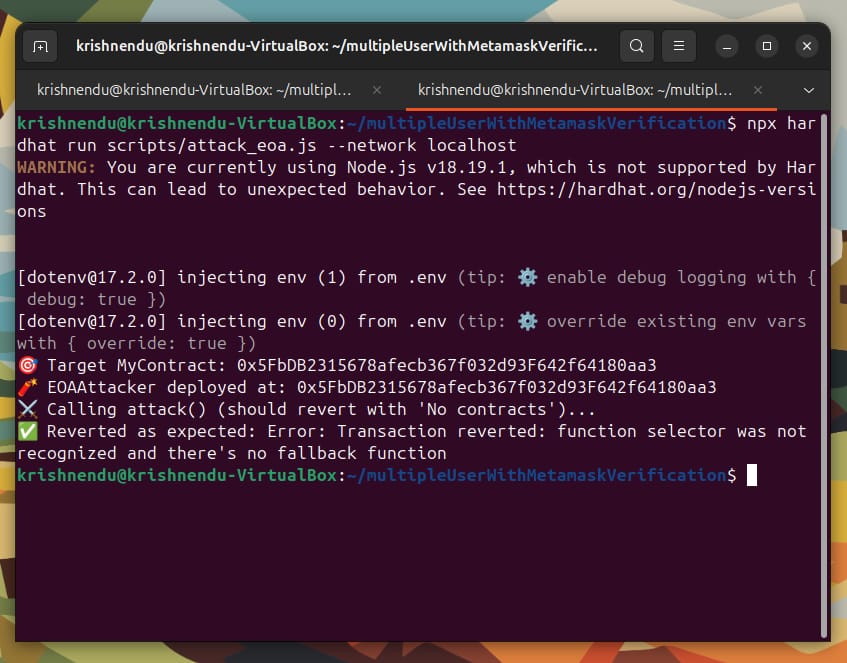
In the above image we have tested **The Reentrancy Attack**

* **Contract Call Instead of Externally Owned Account (EOA) -**

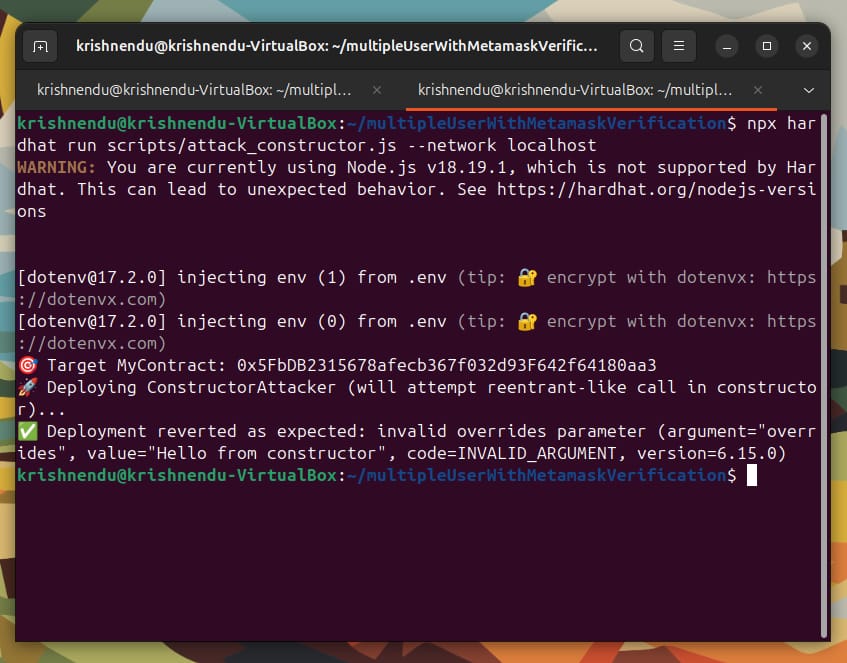
Smart contracts often assume that the caller is an externally owned account (a user wallet). However, other contracts can also call public functions. This can bypass intended rate-limits or access controls. In testing, the setMessage function was invoked through another contract instead of MetaMask. This confirmed that without explicit tx.origin or isContract checks, the system cannot differentiate between EOAs and contracts, leaving it open to automated relay attacks.

We have divided & tested the attack into 3 parts-

1. Contract Call Instead of EOA Attack Variant A - Direct Contract Caller



1. Contract Call Instead of EOA Attack Variant B - Constructor-Based Call

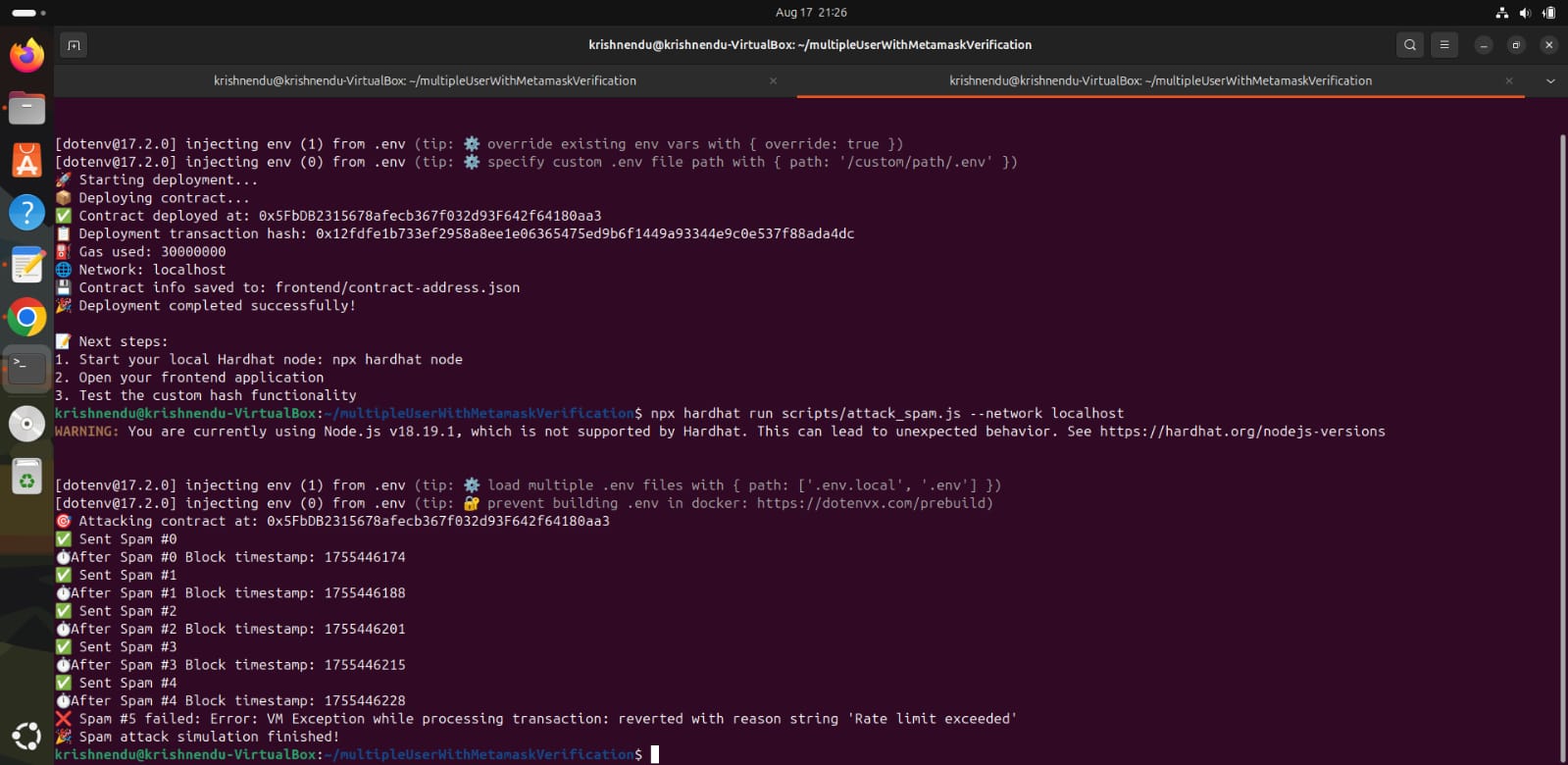
****

1. Contract Call Instead of EOA Attack Variant C - Simple Forwarder (Proxy Call)

****

* **Spam Message Flooding –**

A Denial-of-Service (DoS) style attack was simulated by sending a rapid sequence of large messages in a short time window. Each setMessage call consumes gas and emits an event, bloating the blockchain logs. While the Apothem testnet tolerated the flood until gas limits were exceeded, this showed that malicious actors could fill blocks with meaningless writes, reducing contract availability for genuine users and increasing costs. In the below image we have tested **Spam Message Flooding.**

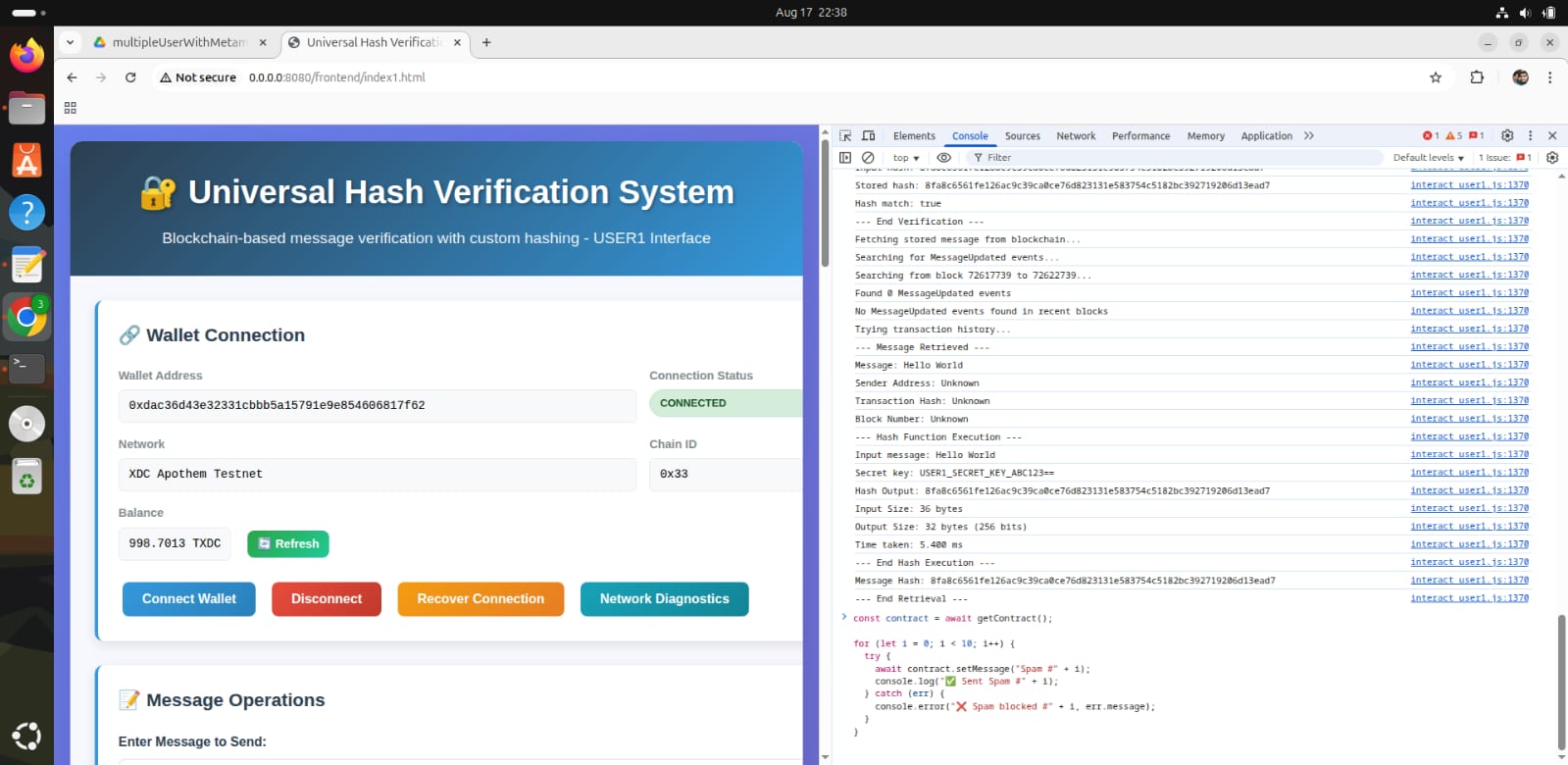
****

* **Null Byte Injection -**

Null bytes (\x00) and other non-printable characters were tested in the setMessage input. Since Solidity does not restrict string content, these were stored and emitted in events without issue. However, in a real DApp frontend, such characters may cause unexpected behavior, break JSON parsers, or enable injection-style exploits. This highlights the importance of sanitizing user input before storage.

**Image:** Testing of Null Byte Injection in Dapp Frontend

* **Rate-Limit Evasion -**

The contract was also tested against rapid repeated submissions to simulate an attacker bypassing message cooldowns. Because the simple design lacked built-in cooldown enforcement or per-user quotas, it accepted consecutive calls without restriction. This demonstrated how, without throttling mechanisms, attackers can overwhelm the system, spam the blockchain, or escalate costs for all participants.

### Local Deployment (Hardhat + Ubuntu)

Once the smart contract was verified in Remix, the next step was **local deployment** using **Hardhat** on an Ubuntu environment.

The procedure followed was:

1. Install Hardhat:

npm install --save-dev hardhat

npx hardhat init

1. Compile the contract:

npx hardhat compile

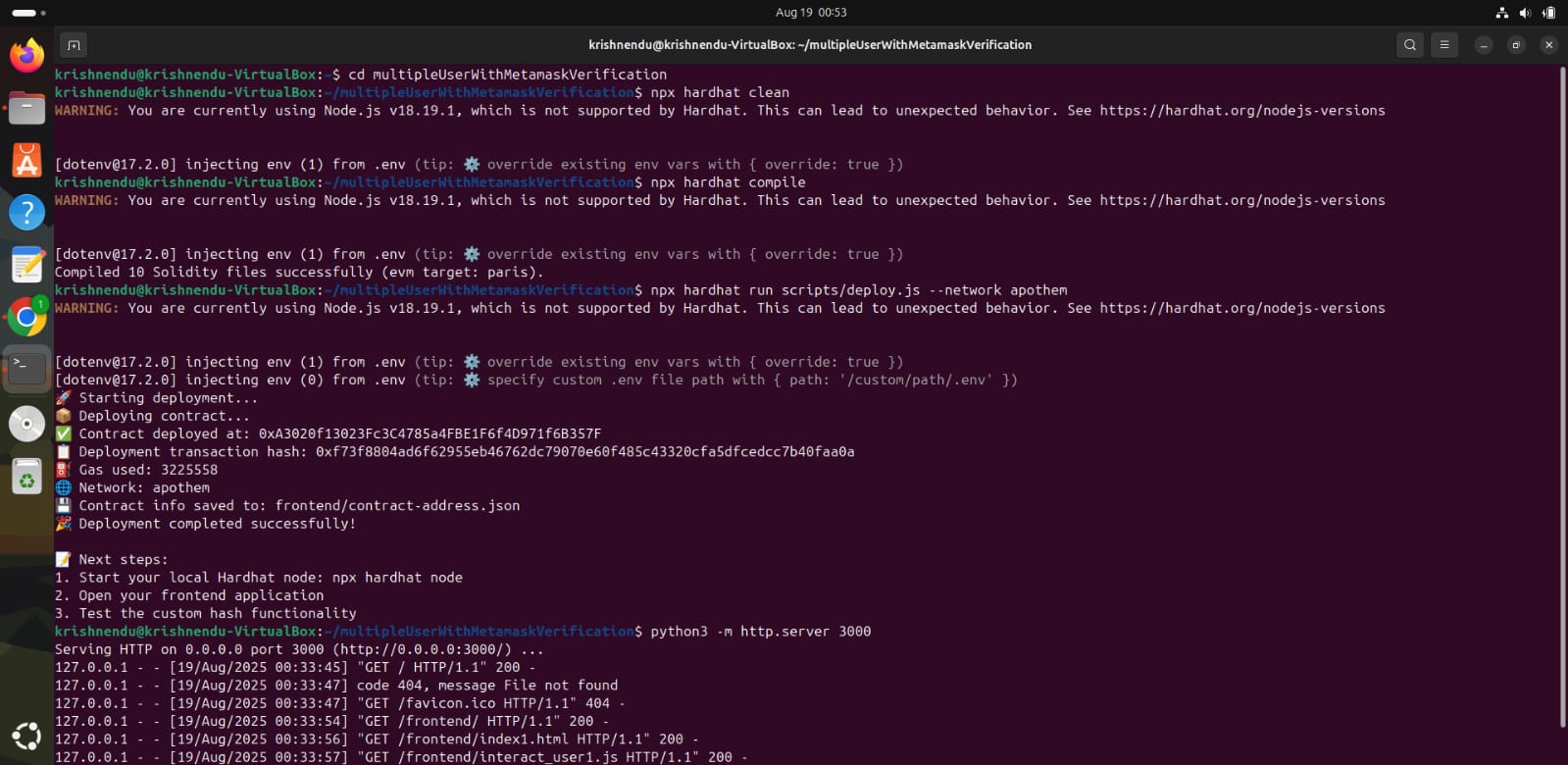
1. Start a local blockchain:

npx hardhat node

1. Deploy the contract:

npx hardhat run scripts/deploy.js --network localhost

After deployment, the contract was tested using the **Hardhat console** and additional JavaScript scripts. Logs confirmed successful deployment and correct behavior of all contract functions.



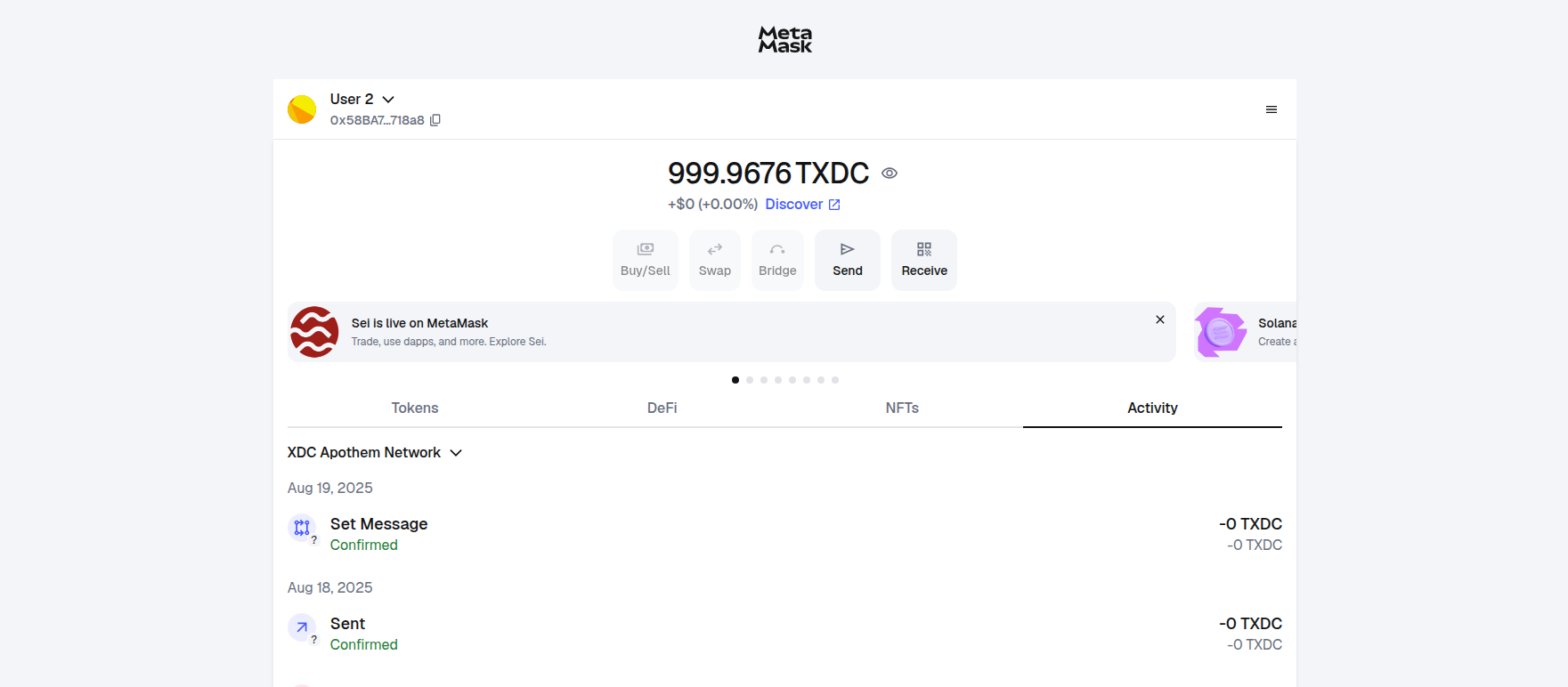
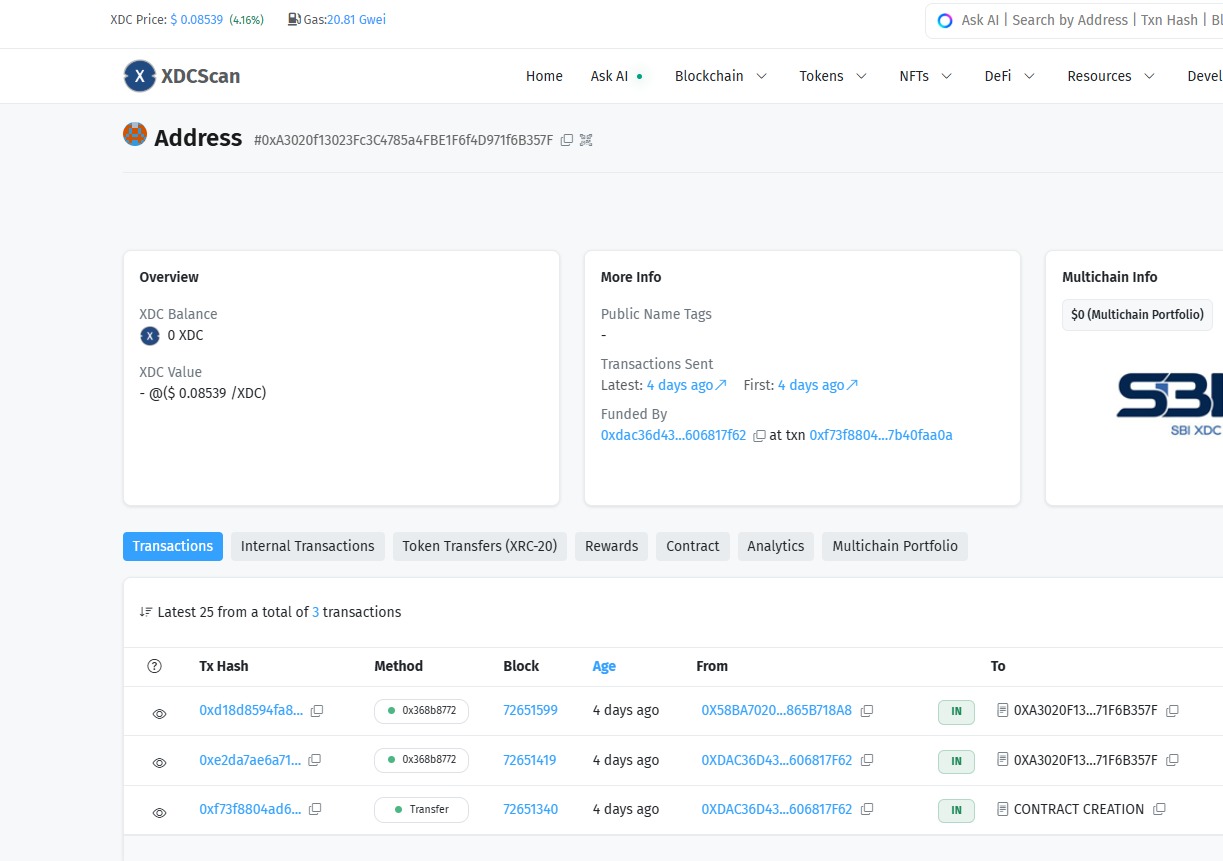
### XDC Apothem Testnet Deployment

Following local deployment, the system was extended to a **public blockchain test environment**. The **XDC Apothem testnet** was selected for this purpose due to its Ethereum Virtual Machine (EVM) compatibility and lower transaction costs.

Steps followed:

* Configured **MetaMask** with Apothem RPC settings and test XDC tokens.
* Updated the **Hardhat configuration file** with Apothem endpoints.
* Deployed the contract using:
* npx hardhat run scripts/deploy.js --network apothem
* Verified the contract on the **Apothem Explorer**.
* Connected the deployed contract to a **React-based DApp**, allowing users to interact with it via MetaMask.

This deployment demonstrated real-world decentralized interaction and validated the integration between smart contracts and MetaMask.



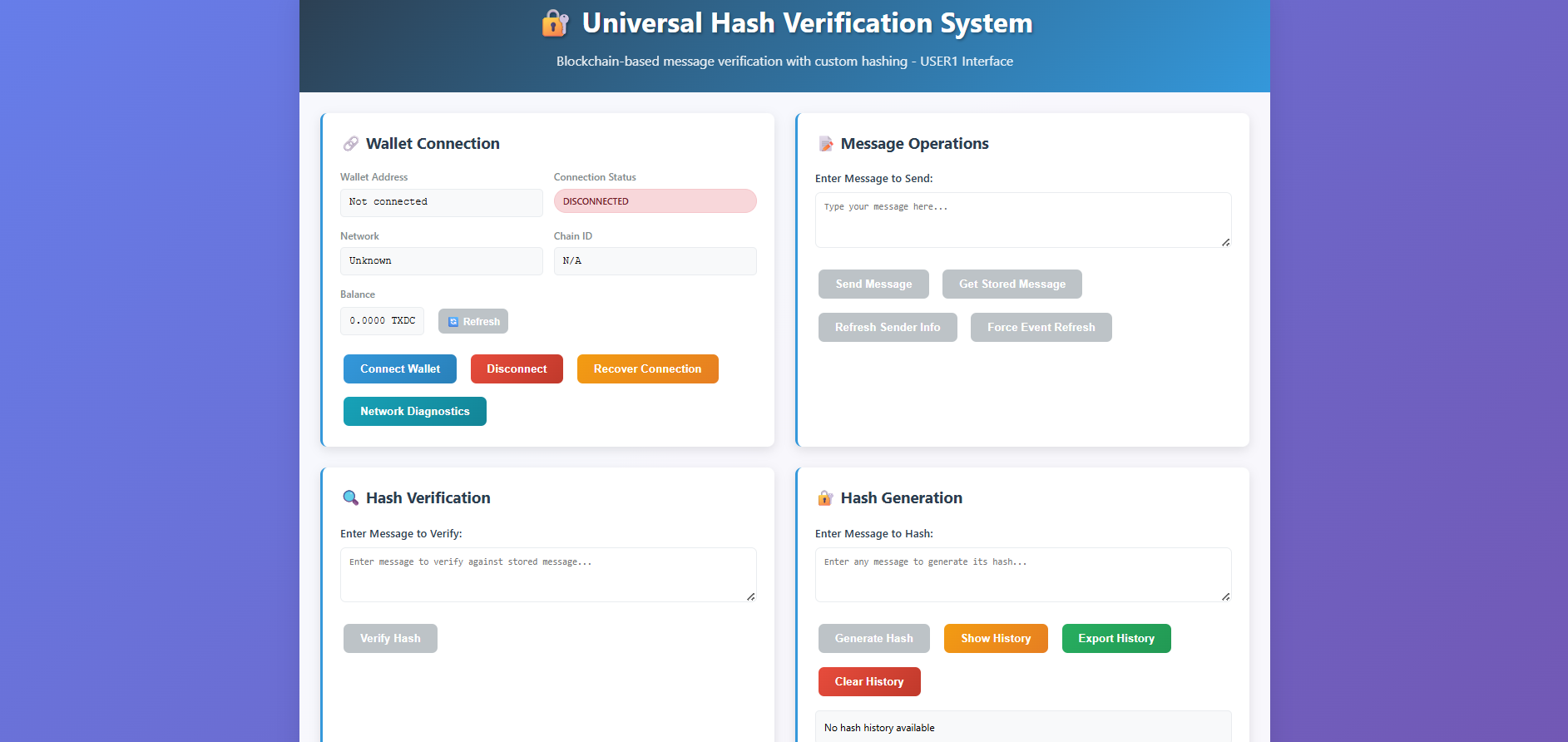
### Custom Hash Integration

The final stage was the **integration of the custom hash function** with the blockchain system.

* **Off-chain hashing**: User inputs were hashed using the custom JavaScript implementation.
* **On-chain storage**: The resulting hash was submitted to the smart contract for permanent storage on the blockchain.
* **Verification process**: During verification, the user resubmits their original data. The system rehashes the input off-chain and compares it with the stored value on-chain. If both match, authentication is confirmed.

This approach reduces blockchain computation costs while ensuring **secure verification** through cryptographic integrity.

After integrating Our Dapp with custom hash function-

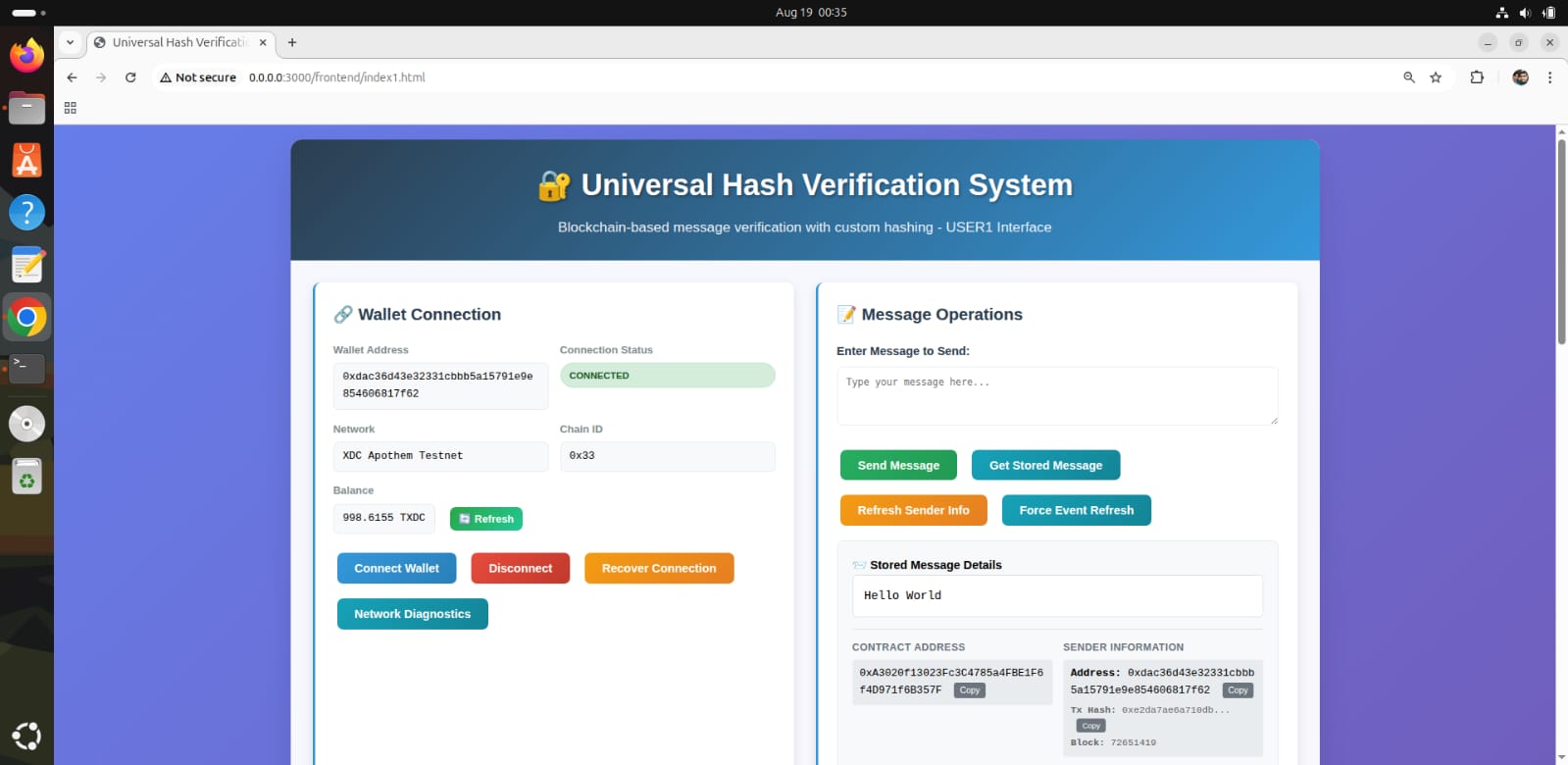
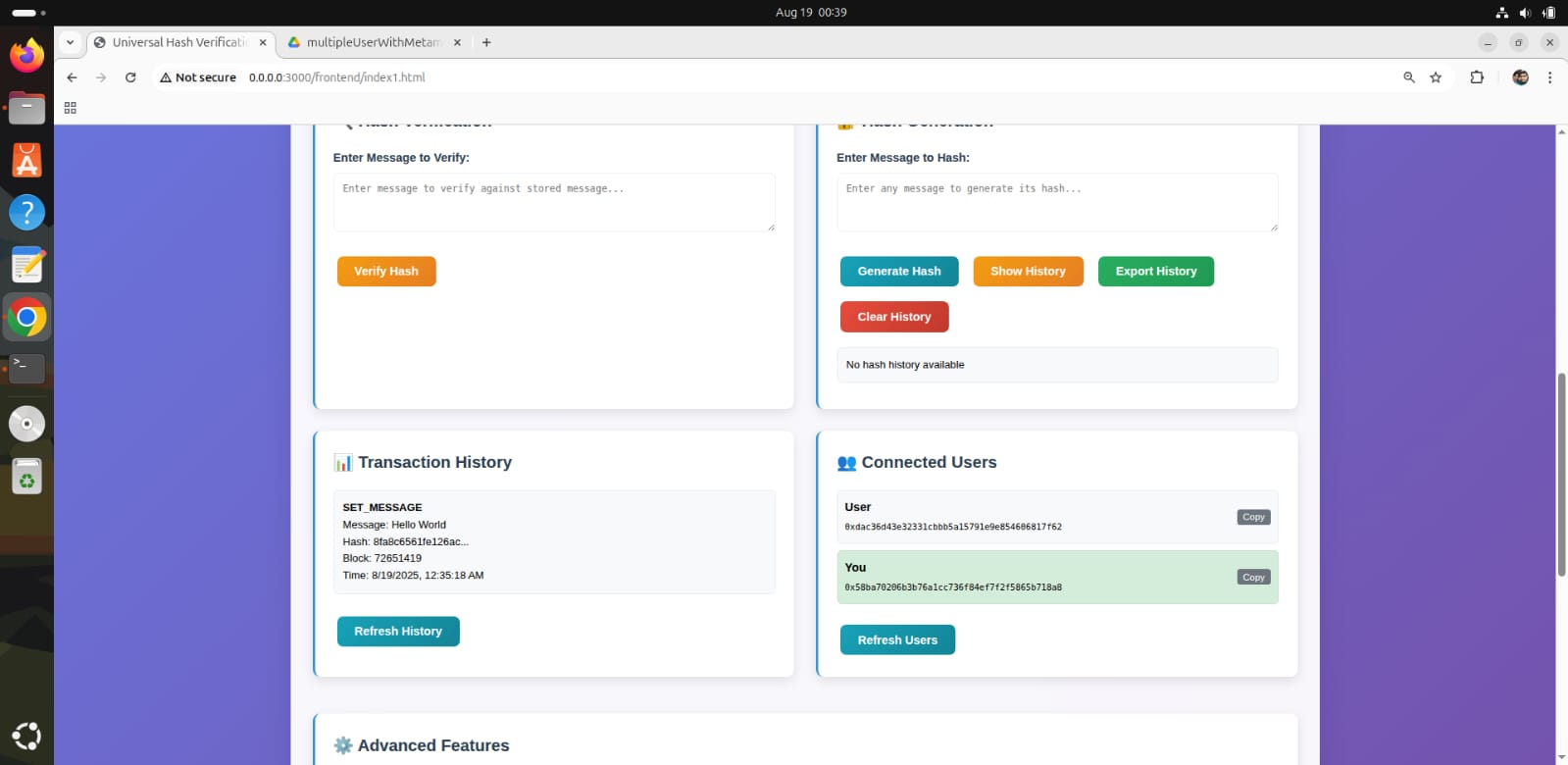


### Frontend DApp Development

To visualize and interact with the entire process, a **frontend decentralized application (DApp)** was developed using **HTML, CSS, and JavaScript**. The frontend provides:

* A user-friendly interface to connect MetaMask.
* Options for registering and verifying user data on-chain.
* Visualization of hash generation and storage steps.
* Real-time interaction with deployed smart contracts on both local Hardhat and XDC Apothem networks.

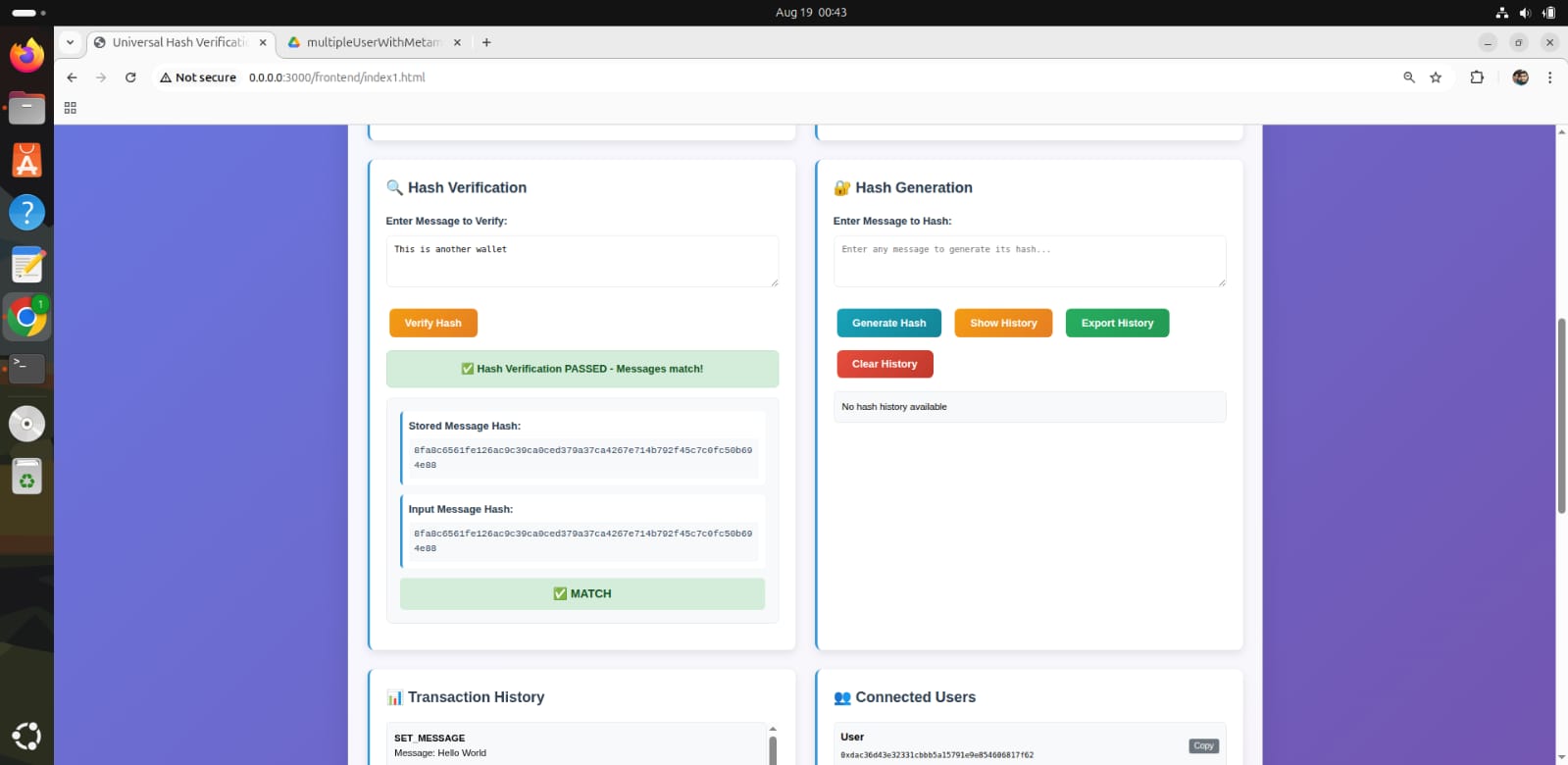
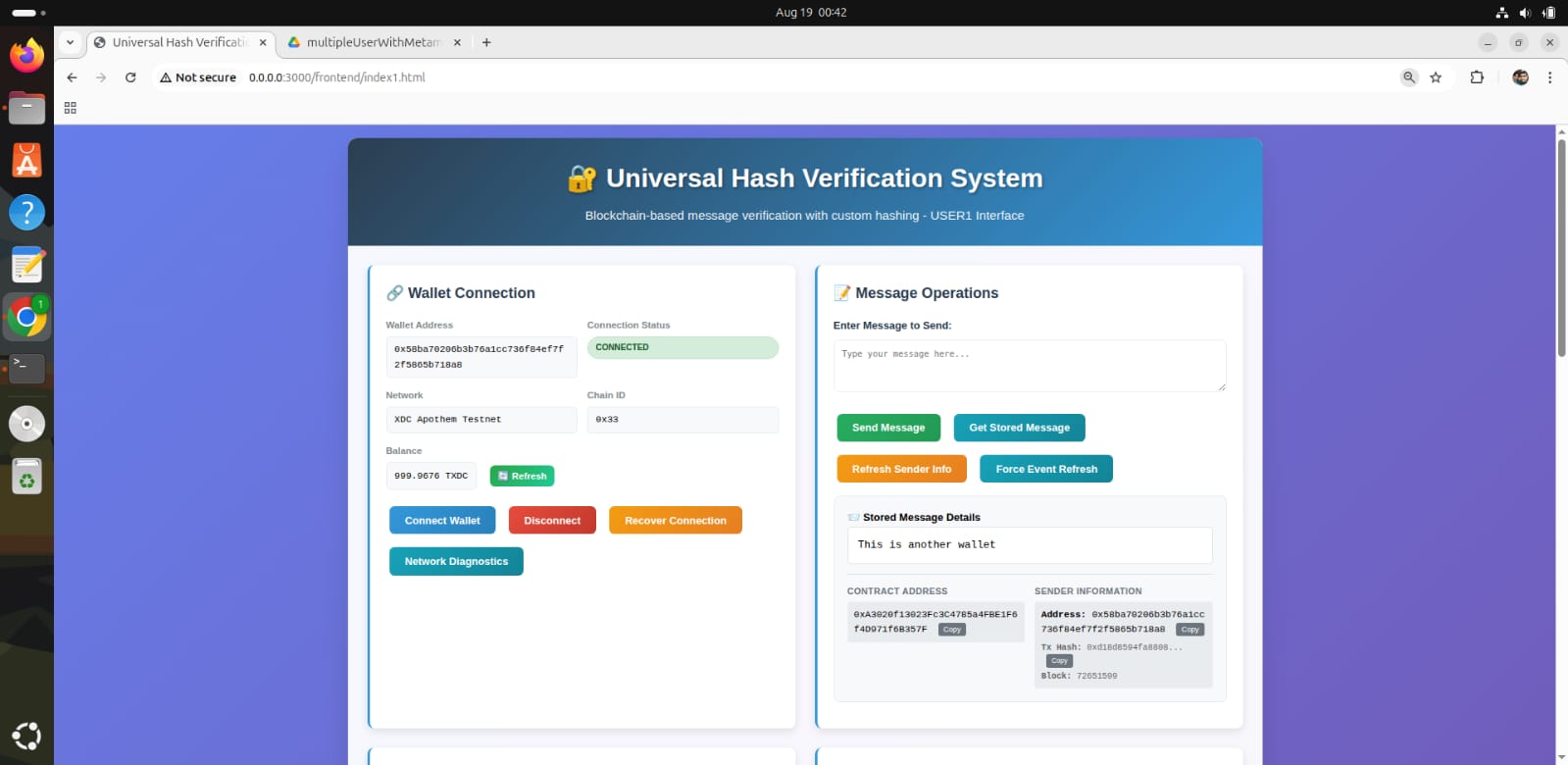
This frontend not only enhanced usability but also demonstrated the complete workflow of the authentication system from a user’s perspective.

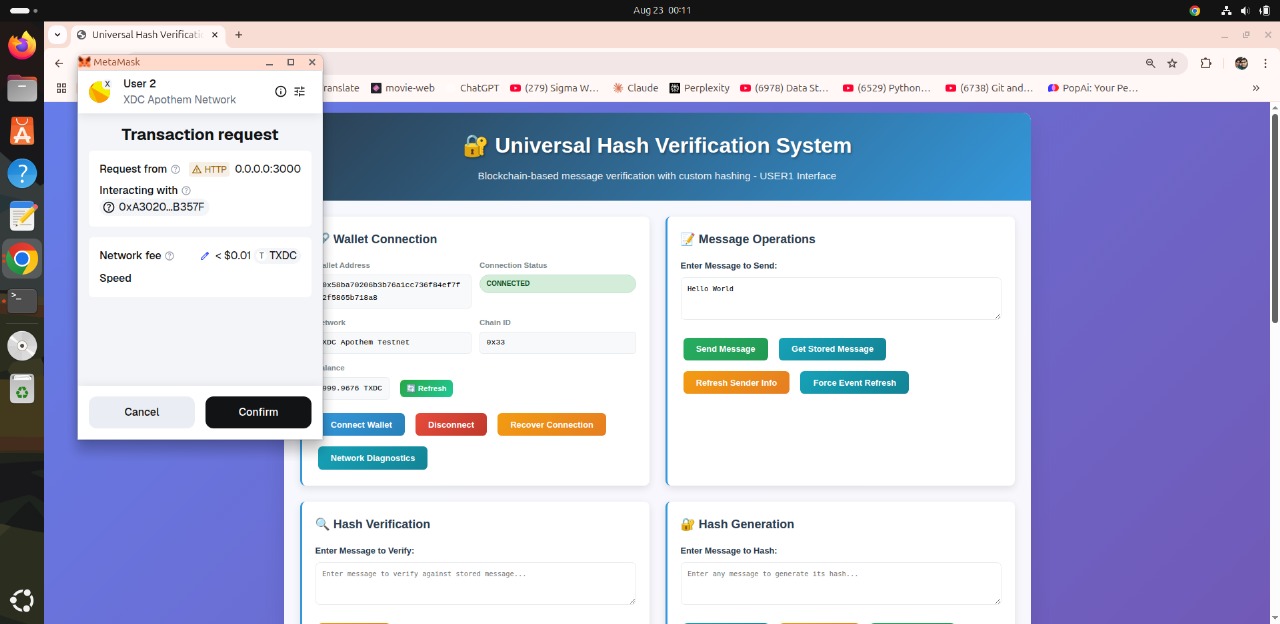


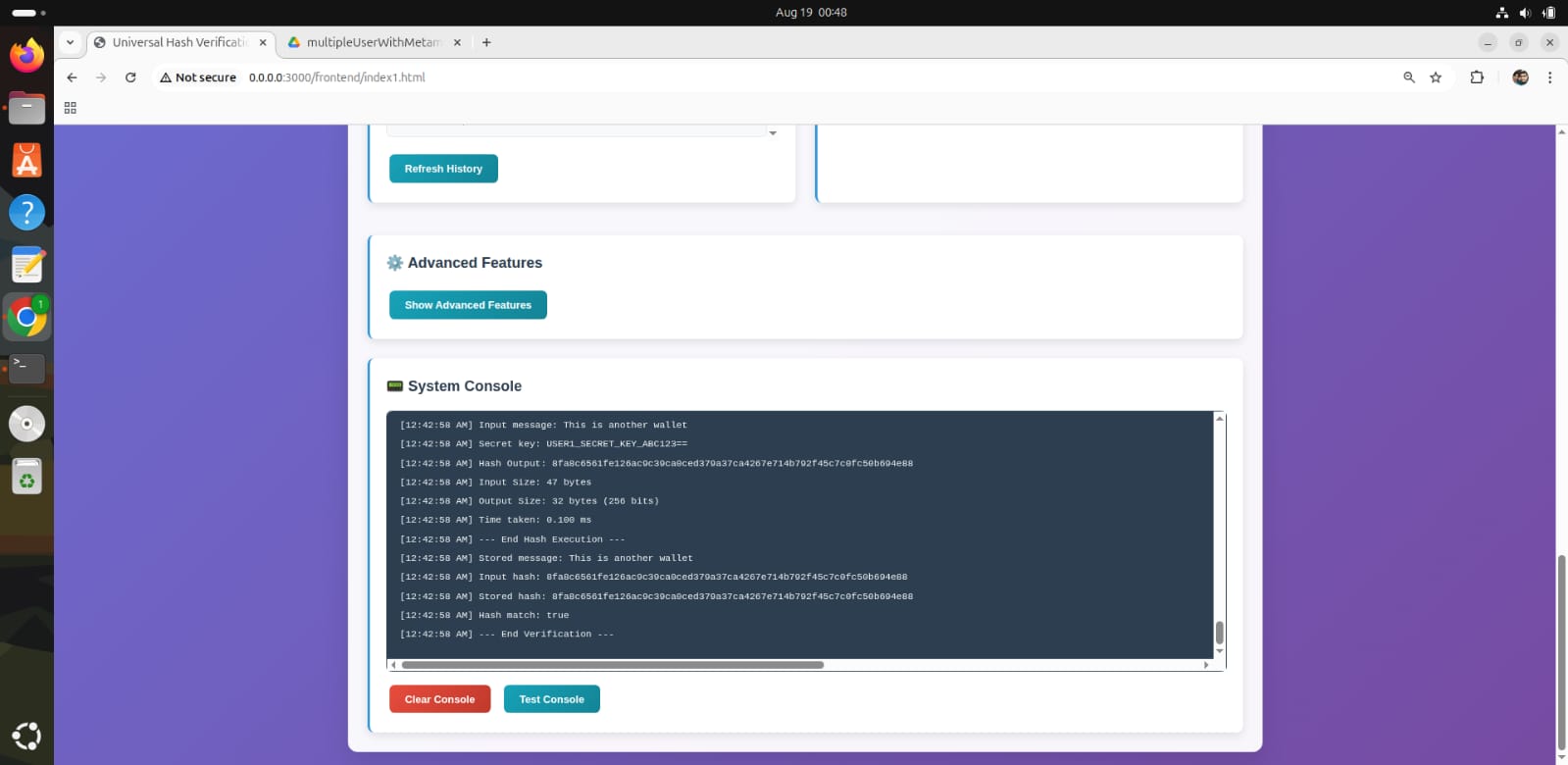
## Results and Observations

The project successfully demonstrates the implementation of a decentralized authentication system that integrates a custom hash function, Solidity smart contracts, and MetaMask interaction. The outcomes and key observations are summarized below:

1. **Hash Function Performance**  
   The custom hash function was benchmarked against standard algorithms such as **SHA-256** and **Keccak (SHA-3)**. While slightly slower than optimized native libraries, it produced consistent fixed-size digests (e.g., 256 bits) and showed stable memory usage. This confirms its suitability for educational and experimental blockchain applications.
2. **Smart Contract Gas Cost**  
   Gas consumption during hash storage and verification was found to be **comparable to conventional hash-based contracts**, indicating that the integration of the custom algorithm did not significantly increase computational overhead on-chain.
3. **Security Testing**  
   Security assessments demonstrated that the smart contract was resistant to multiple common attacks:
   * **Reentrancy attempts** were blocked effectively, preventing repeated unauthorized calls.
   * **Replay attacks** using duplicate signed transactions were rejected.
   * **Arithmetic overflow/underflow** was handled correctly during input validation.
   * **Gas stress tests** confirmed that the system maintained functionality even under heavy load.  
     These outcomes highlight the robustness of the contract design.
4. **Deployment Results**
   * The contract was first deployed **locally on Ubuntu using Hardhat**, where all functionalities executed as expected.
   * A successful deployment was then carried out on the **XDC Apothem Testnet**, integrated with **MetaMask**. Users were able to connect wallets, register, and verify authentication seamlessly through the React-based frontend DApp.
   * Verification on the **Apothem blockchain explorer** confirmed the correctness of deployment.
5. **System Advantages and Challenges**
   * **Advantages**: The system eliminates reliance on centralized credentials, ensures tamper-proof authentication logic, and provides transparency through blockchain immutability. User interaction with MetaMask was smooth, and the DApp offered an intuitive interface.
   * **Challenges**: Limitations include dependence on MetaMask for access, network/gas costs that may affect scalability, and reliance on blockchain availability for consistent performance.

**Overall**, the results confirm that the project successfully achieves secure, decentralized authentication using blockchain, while also integrating a self-designed cryptographic hash function for additional verification. The system proved both **functional and secure**, though real-world adoption would require improvements in **cost efficiency and user accessibility**.





## Future Scope and Limitations

While the project successfully demonstrates decentralized authentication using custom hashing and smart contracts, several limitations and opportunities for further improvement have been identified:

### Limitations

* **Wallet Dependency**: The system currently relies on MetaMask, which restricts accessibility to users familiar with cryptocurrency wallets.
* **Transaction Fees**: High gas costs on Ethereum mainnet may discourage widespread adoption and limit scalability.
* **Connectivity Requirement**: Continuous internet access and blockchain network availability are essential for proper functioning.

### Future Enhancements

* **Cryptanalysis of Custom Hash**: Conduct in-depth cryptanalysis of the custom hash function to evaluate resistance against collision and preimage attacks.
* **Gas Optimization**: Improve smart contract efficiency to reduce costs during large-scale deployments.
* **Layer-2 Integration**: Incorporate Layer-2 scaling solutions (e.g., Polygon, Arbitrum) to minimize transaction fees.
* **Cross-Chain Authentication**: Extend interoperability with other **EVM-compatible blockchains** for multi-chain authentication.
* **Mainnet & Multi-Node Testing**: Deploy on Ethereum/XDC mainnet and test performance in distributed multi-node environments.
* **Enhanced User Interface**: Develop a mobile-friendly and intuitive frontend for better user experience.
* **Role-Based Access Control**: Add enterprise-grade features such as hierarchical permissions and access policies.

## Conclusion

This project can effectively demonstrate how blockchain technology, coupled with MetaMask and smart contracts, can create a secure, transparent, and decentralized authentication system. The system, utilizing a custom hash function and on-chain verification, eliminates centralized servers and is less susceptible to data breaches and stolen credentials.

The Hardhat local deployment and the testnet deployment to the XDC Apothem testnet confirm the solution's validity in production-like blockchain environments. Security testing proved immunity to common attacks, once again validating the approach's resilience.

While challenges such as gas fees, MetaMask dependency, and connectivity requirements remain, the project clearly shows a path forward for decentralized identity management. With the improvement in scalability, cross-chain, and user experience, the system can be an acceptable solution for business and personal authentication needs.