GhostPost: A Pseudonymous Forum Using zk-Promises

Ayhan Mehdiyev
University of Maryland, College Park

Abstract

GhostPost uses the zk-Promises protocol [3] to build an anonymous online forum in which user privacy and accountability are balanced with cryptographic zero-knowledge proofs. This project applies zk-Promises to a client-server setup using React.js, Rust, Python (FastAPI), SQLite3, and Risc0's zkVM framework. By localizing zk-proof generation on the client-side and verification on the server-side, the forum preserves user privacy while enabling moderation through async callbacks to avoid repeated abuse through the enforcement of nullifiers. This paper discusses our design decisions, real-world challenges and solutions.

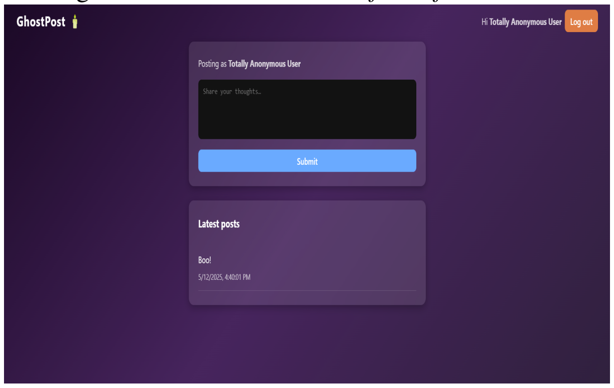
1 Introduction

Anonymous online forums serve a vital purpose of providing open and free communication in a setting in which privacy issues may intimidate honest communication. Holding onto anonymity on the part of users poses serious concerns to moderation, accountability, and integrity of the platform overall. The conventional methods of moderation are heavily reliant on IP and user identity, directly opposing the concepts of user anonymity and privacy.

To solve this intrinsic conflict between ensuring user anonymity and supporting efficient moderation, recent developments in cryptographic techniques hold much promise. The zk-Promises protocol, proposed by Shih et al. (2024), presents a solid system in which zero-knowledge proofs [1] are used to preserve user privacy and achieve accountability via an anonymously updatable state. Users have private states comprising anonymous credentials and ban statuses. These states are kept privately but still publicly verifiable using cryptographic commitments, so moderation does not compromise privacy.

This article introduces GhostPost, a real-world example of a pseudonymous online forum using zk-Promises. Employing a client-server system using React.js, FastAPI based on Python, Rust, SQLite3, and Risc0's zkVM framework,

Figure 1: *Frontend User-Interface of GhostPost*



Boo!
5/12/2025, 4:40:01 PM

GhostPost details a completely operational system in which users anonymously authenticate themselves, post messages, and respond to moderation callbacks without revealing their identities.

2 Background and Related Work

A call to cover fundamental concepts and prior work in zero-knowledge proofs and zk-Promises is necessary to place GhostPost within current research and advancements.

To begin, zero-knowledge proofs are cryptographic structures enabling a party (the prover) to prove knowledge or ownership of certain data to another party (the verifier) without revealing the data itself. The concept was first proposed by Goldwasser, Micali, and Rackoff in 1985 [1] and has been used to a large extent in secure authentication and privacy-centric systems.

Building on the work of zk-proofs, Shih et al. (2024) introduced zk-Promises as a framework tailored uniquely to anonymous moderation and reputation management. The framework involves using zk-proofs together with anonymous credentials to store privately and locally maintained user states. These states are referred to as zk-objects and are privately controlled by the users, but still publicly verifiable through cryptographic commitments. The asynchronous callback system is a notable innovation in zk-Promises through which moderators can force state updates anonymously. Moderation actions, such as banning, are possible asynchronously to maintain moderation efficiency without sacrificing user anonymity.

The real-world instantiation of zk-Promises in GhostPost uses Risc0 zkVM [2], a developer-friendly and low-overhead zero-knowledge virtual machine written in Rust. Risc0 makes zk-proof creation and verification easier by providing a user-friendly Rust-oriented development model, which improves development simplicity and execution performance significantly. GhostPost also uses common technologies to aid in its implementation. SQLite3 offers a light but robust relational database management system best suited to the bulletin board design that zk-Promises needs. To build high-performance APIs, Python's framework, FastAPI, is utilized in the backend because it accommodates asynchronous operations and effortless integration with WebSockets.

Lastly, GhostPost incorporates WebSockets to facilitate smooth real-time communication between the client and server. The duplex communication of WebSocket outpaces conventional HTTP polling and provides remarkable performance advantages in applications with high latency sensitivity.

Together, all these concepts and technologies make GhostPost a pragmatic and innovative example of a privacy-respecting and responsible online system.

3 System Specifications

The GhostPost system relies on specific versions and dependencies of software to maintain compatibility, stability, and performance. Since we use a wide range of technologies in this stack, our decisions were made primarily for compatibility reasons:

- Rust: Version 1.77+ was used because it offers remarkable performance, guarantees of memory safety, and strong concurrency support, which are essential for zk-proof computations to work efficiently.

- Risc0 zkVM: Version 1.2.4 was used as per the specifications outlined by Shih et al. (2024). Risc0 provides a simple interface to generate and verify zk-proofs using Rust, streamlining the process and making it less complicated.

- React.js, Vite, and Axios: The most recent and stable versions were used due to their up-to-date functionality. React.js enables a modular and maintainable frontend architecture. Vite provides fast development and optimized bundling. Axios allows us to build seamless HTTP and WebSocket communication between the client and server systems.

- Python FastAPI, WebSocket, and CORSMiddleware: The most recent and stable versions were used due to their high performance and simplicity of use. FastAPI was chosen based on its ease of building RESTful APIs with low overhead and support for asynchronous operations to efficiently manage multiple WebSocket connections. CORSMiddleware was needed to manage cross-origin requests safely.

- SQLite3: The most recent and stable version was used due to its light footprint and dependability. Both its file-accessible design and low-resource consumption make it perfect for small to medium-sized programs like GhostPost to have persistent storage without excessive overhead or added complexity.

Collectively, these technologies present a balanced mixture of performance, scalability, development simplicity, and integration features required to introduce a strong, efficient, and privacy-protecting forum building on zk-promises.

4 System Architecture

GhostPost has a modular client-server design tailored to privacy-preserving interactions and moderation using zero-knowledge proofs. The design is broken down into three main layers: the frontend client application, the backend API server, and the zk-proof engine, each backed by a structured file system to make maintenance and scaling easier.

The root of the structure consists of three main directories:

- **frontend/** –Holds the client-side React.js + Vite application responsible for the user interface and interactions.

- **backend/** –Serves the Python FastAPI server, responsible for handling API routes, WebSocket connections, and database management.

- **zk-simple/** –Enacts the Rust zk-proof engine using the Risc0 zkVM to perform cryptographic proof creation and verification.

4.1 Frontend Client Application

The frontend is developed using React.js with Vite to give a user-friendly and responsive interface to register, log in, and post on the forum. The communication between the frontend and the backend is taken care of using Axios to fetch data and update data in real-time through WebSocket integration.

4.2 Backend API Sever

The backend uses FastAPI, which was selected because it has efficient asynchronous execution and robust integration features with WebSockets to support real-time interactions. It handles incoming requests, verifies zk-proofs received from the clients, and all database operations are facilitated through concise routes.

4.3 Zero-Knowledge Proof Engine

The zk-proof engine plays a core role in ensuring anonymity of users and compliance with moderation policies. Developed on Rust and Risc0 zkVM, it creates and validates zero-knowledge proofs while keeping user privacy intact but making state transitions accountable.

4.4 UserState (zk-object)

The private user object maintained on the client-side includes vital information on anonymity and moderation:

```
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct UserObject {
    pub is_banned: bool,
    pub tickets: Vec<u128>,
    pub current_internal_nonce: u128,
}
```

`is_banned` : Whether or not a user is banned
`tickets` : Stores tickets on behalf of user posts
`current_internal_nonce` : Current random serial

4.5 Database Layer

The SQLite3 database contains separate tables designed to handle user information, forum posts, and cryptographic commitments optimally:

```
// Users Table
id: Integer (auto_increment constraint)
username: String (unique)
password_hash: String

// Posts Table
id: Integer (primary key)
content: Text (message board posting content)
timestamp: DateTime (creation time)
ticket: String (used in callbacks for moderation)

// Commitments Table
id: Integer (primary key)
commitment_hash: String (unique hash of UserObject)
nonce: String (old nonce, anti-replay protection)
new_ticket: String (ticket of most recent post)
created_at: String (audit trail timestamp)
```

```
// Callbacks Table
id: Integer (primary key)
ticket: String (removed post's ticket)
action: String (moderation action, i.e., "ban")
created_at: String (audit trail timestamp)
```

This organized design unambiguously separates concerns, ensuring maintainability, scalability, and efficient integration of the varied layers of the GhostPost system.

5 Implementation and Design Decisions

This section offers an in-depth look at the design decisions and issues faced while designing GhostPost, including managing zk-proofs, frontend and UI aspects, and back-end design considerations.

Originally, for ease of development, GhostPost would make a request to the server in order to generate the zk-proof. This was done because of the complexity of invoking Rust through React.js compared to the ease of doing so within Python. However, this had a serious and fundamental privacy problem because the server would have to have access to the sensitive data of users. To overcome this, attempts were made to migrate the zk-proof generation completely to the client-side. Numerous solutions were considered:

- Client-side Process Spawning: One of the first things tried was spawning client-side processes separately. This was unsuccessful because of the strict cross-origin policies enforced by modern web browsers.
- Electron Experimentation: The Electron framework was considered as a possible alternative. While Electron tackled the cross-origin problem, it added a lot of overhead and complexity at the expense of user experience and maintainability.
- Custom Vite Plugin: The ultimate and successful resolution was the creation of a custom Vite plugin. The plugin allowed fast and safe client-side generation of zk-proofs in the React frontend.

Vite was the choice instead of options like WebAssembly (WASM)[1] because of its fast build cycle, simplicity of integration with React, and low overhead. This allowed development to work much more straightforwardly; it maintained security by having critical operations on the client-side, and it bypassed technical challenges. Data sent to `prove.rs`, like callback tickets as well as user object information, was intentionally selected to include all required context to verify without unnecessary server communication. Also in `verify.rs`, a serializable replica of the Journal structure was used, mapping `u128` types to strings. The mapping took care of precision issues found in JavaScript, namely in React's state management and handling of numbers. Strings gave a safe and problem-free way of handling numeric data between JavaScript and Rust boundaries, even if it somewhat added to serialization complexities.

The frontend development used a React-based design with an emphasis on simplicity and ease of development. An explicit choice was to avoid a heavy modularization of the frontend into independent components to minimize complexity and accelerate the early stages of development. The simplified design allowed fast prototyping and testing and facilitated quick iteration cycles. Persistent login/logout states were implemented directly into the main App component to make it easier to manage sessions and states. UX considerations were carefully made to include clean ban notices, persistent session management, and elegant user messaging to keep users properly guided through forum interactions.

WebSocket was used for real-time two-way communication because it is efficient, has low latency, and outperforms the conventional HTTP polling. This allowed it to provide instant and responsive communication essential for forum operations.

Furthermore, the limit on a single post per session was implemented as a deliberate design to uphold rigorous moderation control and state management simplicity in zk-proof protocols to boost both moderation integrity and security.

The backend used Python's FastAPI for quick development and strong asynchronous request processing. CORSMiddleware was used to safely handle cross-origin requests, essential to achieve stable interactions between the frontend and backend services.

WebSocket integration in FastAPI was used to supplement the frontend's requirements of real-time communication by enabling smooth bidirectional data exchange. This configuration lowered server overhead and maximized application responsiveness.

For database design purposes, a single SQLite database was used instead of having different databases for forum functions and zk-specific data. This kept the design much simpler overall and streamlined database interactions to minimize synchronization problems and make maintenance easier.

The final decision that may raise eyebrows is our empty `__init__.py` files, which we kept in Python directories to explicitly mark Python modules and to maintain proper structuring and readability of the package. Collectively, these intentional and strategic design choices all contributed to a unified, safe, and sustainable implementation, optimally balancing efficiency, complexity, and user privacy.

6 zk-Promise Implementations and Challenges

At the core of GhostPost is the zk-Promises framework for supporting anonymous but accountable user state changes. We centered our work on client-stored zk-objects, callbacks from the server, and zero-knowledge state commits with the Risc0 zkVM. This section describes how we developed, implemented, and rationalized each of the key parts of the protocol from both the technical constraints of zero-knowledge systems and our target application goals.

6.1 Callback Mechanism and Asynchronous Enforcement

Our system enables the moderator to apply asynchronous punishments using "callbacks" (server commands bound to post tickets). The client retrieves all callbacks from the bulletin board when logging in and matches them against the local list of tickets maintained by the user. In case of a match, the user is marked as banned in their zk-object. For the sake of simplicity, we kept it at one callback for a ban, but kept the design open to scale for future actions, such as warnings and reduction in reputation. The enforcement mechanism is directly implemented in the zero-knowledge circuit (`main.rs` in `methods/guest`). The inner loop on `user_object.tickets` and `callback_tickets` enforces accountability while keeping the flagged post a secret. In case of finding any match, the state is set to banned, and the circuit aborts with an error if the user tries to authenticate when banned. This design follows the asynchronous moderation model specified in the zk-Promises paper (Shih et al., 2024), with callback enforcement occurring during events caused by users like login and not directly on violation.

6.2 State Transition and Nullifier Logic

The `PrivateInput` type contains the complete context required for a safe zk-object transition in the form of the user's current state (`UserObject`), the old and new nonces, and the server's signature on the commitment. The type is passed into the guest circuit via the Risc0 context. A fundamental design choice was mandating server-side authentication of the purported commitment hash. This is implemented using ECDSA signature verification, identical to the scheme outlined by the zk-Promises spec. This stops users from forging previous states or providing fake commitments. To safeguard against replays and guarantee atomic transitions, every update discloses the previous nonce (`old_nonce`) and commits to a fresh state (`new_commitment`) using a new `new_nonce`. This serial number (nonce) mechanism is analogous to the nullifier-based protections found in zk-Promises.

6.3 Why a Simple Vector was Selected for Tickets

As opposed to using Merkle trees or accumulator-based callbacks in the zk-Promises paper, we used a straightforward `Vec<u128>` to keep track of user tickets. Although less space, such a structure is better suited to circuits, has dynamic length capability, and sidesteps the algebraic overhead of proving set inclusion in zero-knowledge. Our proof sizes were low, and our performance was acceptable in our use case of a forum.

As our call list was public but also short (a couple dozen active posts per user), privacy threats associated with linear scanning were minimal. This sacrifice enabled quicker development while staying consistent with the zk-Promises model in spirit.

6.4 Accuracy and Serialization in Proof Checking

The most important implementation hurdle came from the inability of JavaScript to deal with 64-bit and 128-bit integers. Since React and the majority of modern web browsers cannot safely stringify or parse `u128` values without losing precision[2], we had to convert all of them to strings when communicating with the frontend.

This choice is seen in `verify.rs` as the public Journal struct (output of the zk-proof) is mirrored to `JournalOut`, representing each `u128` as a decimal string to safely serialize to JSON. The same is done on the front-end and in `prove.rs` with all the callbacks and nonces treated as strings to maintain compatibility.

This was a workaround to maintain cross-platform safety while sacrificing nothing in terms of proof correctness. Lower-level frontend libraries (e.g., `Rust/WASM`) could obviate it, but React's maturity and developer ergonomics were a higher-priority concern.

6.5 zk-Proof Input Design and Safety Verifications

In `prove.rs`, the proof input is organized as a JSON file (`ProofRequest`) received from the backend and contains:

- Current user tickets (as strings)
- Callback tickets (as strings)
- Old commitment nonce
- Banned flag (of the past login)

We convert and simulate activity all tickets and nonces to `u128`, and mint a fresh ticket prior to proof generation. The fresh ticket is included in the user's ticket list and used as a foundation to build the next session. Also, two dummy tickets are added if the user is inactive to keep the zk-object intact[3]. This design was initially thought to be unnecessary but was maintained to avoid risks and to maintain compatibility with all stages of development.

6.6 Reflection on zk-Promises Goals

In total, our design meets the main objectives of zk-Promises:

- Local-held state: zk-objects are created and maintained locally.
- Asynchronous moderation: callbacks are used at login.
- Zero-knowledge enforcement: user bans and state changes are established privately.
- Replay protection: nullifiers avoid proof reuse.
- Proof generation in the client and proof verification in the server.

We pragmatically adopted these concepts in favor of serial number-based replay protection, linear scanning, and low-complexity circuits at the expense of full Merkle logic to optimize for clarity, modularity, and implementability in a classroom and forum context. Our system demonstrates that zk-Promises are deployable into practical applications even with minimal cryptographic infrastructure and using a simple-to-implement design strategy.

7 Discussion and Future Work

While GhostPost successfully demonstrates a functioning zk-promises-based forum, several limitations and future enhancements are worth considering:

- Scalability: Although the current implementation is functional, it is in fact limited to small-scale forums. The use of vectors for ticket scanning, in particular, amongst other things, does not scale well to hundreds or thousands of posts.
- Centralized Backend: We still have to partly trust the backend.
- Frontend: We had to introduce extra conversions and complexity to account for JavaScript's number handling limitations, which won't scale well and could very quickly lead to technical debt.

That being said, the future of GhostPost does look bright:

- Decentralized Bulletin Board: Migrating from a centralized SQLite database to a blockchain would allow fully trustless moderation.
- Merkle Tree Proofs: Integrating Merkle Proofs would reduce circuit size and improve privacy.
- Server Keys: Currently, a fresh ECDSA keypair is generated per proof session, ideally, we would improve security with persisting keys across sessions.

- Multi-Post Sessions: Although it would require more complex client-side proof generation and session handling, future iterations could allow for multiple posts per session.
- Advanced UserObjects: Currently, we only keep track of the bare bones minimum needed to implement zk-Promises. In the future, we could implement reputation and warnings.
- UI Componentization: Although a single frontend served our purposes quite well, modular React components would improve maintainability, scalability, and testing by a lot.

8 Conclusions

GhostPost addressed many technical challenges, and through careful design decisions, we achieved a lightweight yet functional prototype. This project is a testament that the goals of zk-Promises can be realized in real and much larger applications.

9 Acknowledgments

We would like to thank the CMSC498C teaching staff for their guidance, as well as the authors of the zk-promises paper (Shih et al., 2024), whose framework formed the backbone of our implementation. We also acknowledge insights and clarifications received from Piazza discussions throughout the project development.

References

[1] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems.

[2] INC., R. Z. Risc zero zkvm: A general-purpose zero-knowledge virtual machine. Whitepaper, 2023.

[3] SHIH, M., ROSENBERG, M., KAILAD, H., AND MIERS, I. zk-promises: Anonymous moderation, reputation, and blocking from anonymous credentials with callbacks. Cryptology ePrint Archive, Paper 2024/1260, 2024.

Notes

[1]WebAssembly was considered, but Vite offered easier integration with React and a faster development cycle.

[2]JavaScript uses 64-bit floats, which cannot represent all 128-bit integers precisely.

[3]Originally added to prevent prover failures during early development. Retained as a safeguard.