Verifiable Gacha System with Zero-Knowledge Proofs

Yanze Lyu

Repo Location: https://github.com/SWYZJustin/BangDream-Zero-Knowledge-Gacha

1. Introduction and Motivation

Randomized reward systems (commonly known as gacha or loot-box mechanisms) are widely used in modern games and online services. Although these systems advertise fixed probability distributions and guarantee mechanisms, users have no direct way to verify that the server actually follows the stated rules. In practice, all randomness and reward computation are performed server-side, which requires users to trust that the server does not manipulate outcomes.

The goal of this project is to explore whether zero-knowledge proofs can be used to make such randomized systems verifiable without revealing sensitive internal state. I implemented a web-based gacha application in which the server generates random draws and produces cryptographic proofs that the draws were computed honestly, while clients verify these proofs locally. The system demonstrates how modern SNARK systems can enforce fairness properties such as correctness in randomness, pity guarantees, and more, in an application-level setting.

2. System Overview

The system follows a client–server architecture. The server is responsible for generating random seeds, computing gacha outcomes, and producing zero-knowledge proofs. The client verifies these proofs and only accepts results that pass verification.

The application is implemented as a Next.js web application with server-side API routes. Zero-knowledge proofs are constructed using Halo2, a PLONK-based universal SNARK. The proving logic is written in Rust, and the verification logic is compiled to WebAssembly so that proofs can be verified directly in the browser.

From the user's perspective, the application behaves like a standard gacha system: the user selects a pool and performs a single draw or a ten-draw. Internally, each draw is accompanied by a proof that certifies the correctness of the random generation, rarity computation, and any guarantee logic.

3. Zero-Knowledge Circuit Design

The core of the system consists of several Halo2 circuits that model different gacha behaviors. I implemented four circuits: single draw, single draw with pity, ten draw, and ten draw with pity.

Randomness is generated using a Linear Congruential Generator (LCG). While LCGs are not cryptographically secure by themselves, the seed is treated as a private witness and never revealed to the client. The circuit enforces that the published random values are consistent with the LCG transition rules, ensuring that the server cannot arbitrarily choose outcomes after the fact.

Rarity is computed by mapping the random value modulo a fixed range into discrete rarity tiers using threshold comparisons. These comparisons are enforced inside the circuit using boolean flags and arithmetic constraints, ensuring that the published rarity is consistent with the advertised probability distribution.

For circuits with a pity mechanism, the circuit additionally enforces conditional logic based on a public streak counter. When the number of consecutive non–high-rarity draws exceeds a fixed threshold, the circuit forces the final rarity to the highest tier and resets the streak counter. Otherwise, the streak counter is incremented. This logic is fully encoded as arithmetic constraints, so the server cannot bypass or selectively apply the pity rule.

4. Proof Generation and Verification Workflow

For each draw request, the server performs the following steps. First, it samples a fresh random seed, which serves as the private witness. Second, it computes the random value(s), rarity outcome(s), and updated streak counter (if applicable). Third, it generates a zero-knowledge proof attesting that all these values were computed according to the circuit constraints.

The server returns the public outputs together with the proof to the client. On the client side, the browser loads a WebAssembly verification module and checks the proof using the corresponding verification key. Only if verification succeeds does the client display the result to the user.

This workflow ensures that trust is shifted from the server's honesty to the soundness of the cryptographic proof system. A malicious server that attempts to manipulate probabilities or skip guarantee logic would be unable to produce a valid proof.

More Details of the workflow:

—----------

Random Seed and LCG Generation
The server generates a fresh random seed, which is treated as a private witness and never revealed.

Random values are derived using a Linear Congruential Generator (LCG).

LCG transition constraint:
a * input + c = output + m * q

where:
- input is the seed or previous random value,
- output is the next random value,
- a is the multiplier,
- c is the increment,
- m = 2^32 is the modulus,
- q is an auxiliary quotient.

This constraint ensures that each random value is correctly derived modulo 2^32.

For ten-draw circuits, this constraint is applied sequentially, enforcing that each output feeds into the next LCG step.

—----------

Modulo Reduction for Rarity Mapping
To derive rarity, the circuit computes a bounded value:

random = random_mod_10000 + 10000 * q_10000

This enforces:
random_mod_10000 = random % 10000

and ensures that rarity computation depends only on the lower 4 decimal digits of the random value.

—----------

Rarity Selection Constraints
Rarity is constrained to take exactly one of the valid values {1, 2, 3, 4} using:

(rarity - 1)(rarity - 2)(rarity - 3)(rarity - 4) = 0

Threshold-based selection is enforced using binary indicator flags:

flag_5 + flag_4 + flag_3 + flag_2 = 1

Each flag activates a corresponding range constraint on random_mod_10000, ensuring that exactly one rarity tier is selected and that it matches the predefined probability intervals.

All flags are constrained to be binary:
b * (b - 1) = 0

Pity Activation Logic
For pity-enabled circuits, the pity condition depends on the public streak counter.

A binary variable pity_active indicates whether the pity threshold has been reached.

Binary constraint:
pity_active * (pity_active - 1) = 0

Pity enforcement constraints:
pity_active * (final_rarity - 4) = 0
(1 - pity_active) * (final_rarity - computed_rarity) = 0

These ensure that when pity is active, the final rarity is forced to 5★, and otherwise the computed rarity is used.

Streak Counter Update
The streak counter update depends on whether the final rarity is 5★.

A binary indicator is_5star is enforced:
is_5star * (is_5star - 1) = 0

Streak update constraints:
is_5star * streak_next = 0
(1 - is_5star) * (streak_next - streak_prev - 1) = 0

This ensures that the streak resets to zero after a 5★ draw and increments by one otherwise.

Client-Side Verification
The client receives the public outputs (random values, rarities, streak_next if applicable) and the zero-knowledge proof.

The proof is verified using the corresponding verification key. Successful verification implies that all constraints above were satisfied, and the gacha result is accepted by the client.

More details can be found in the application -> ZK Flow

5. Security Properties and Limitations

The system provides several security guarantees. First, it enforces computational integrity: the server cannot publish outcomes that do not follow the specified random generation and rarity computation rules. Second, it preserves privacy: the random seed remains hidden, and only the derived outputs are revealed. Third, the pity mechanism is cryptographically enforced, preventing selective application or manipulation.

There are also limitations. The system does not guarantee unpredictability of randomness in a cryptographic sense, since the LCG is not a secure pseudorandom generator. Instead, the focus is on verifiability and non-manipulation after the seed is chosen. In addition, proof generation is computationally expensive compared to traditional server-side logic, which limits scalability without further optimization or batching.

Appendix. How to Run the System

The application can be executed locally or accessed through a deployed Vercel demo.

Local Setup

Prerequisites:
- Node.js
- Rust toolchain
- wasm-pack
- Rust target: wasm32-unknown-unknown

Build Steps:

1. Install JavaScript dependencies:
   yarn install

2. Build the WebAssembly modules for the Halo2 circuits:
   yarn build:wasm

3. Build the Next.js application:
   yarn build

4. Run the development server:
   yarn dev

The application will be available at:
http://localhost:3000

Zero-knowledge proofs are generated on the server side, and proof verification is performed in the browser using WebAssembly.

Vercel Deployment

A live version of the application is deployed on Vercel using its serverless platform.
Each deployment rebuilds the Rust circuits, compiles them to WebAssembly, and deploys the Next.js application.

The deployed version runs the same proof generation and verification logic as the local setup and serves as an interactive demo of the system.
