A Differentially-Private Implementation of Voting System

Dang Dinh, *Boston University* Yixin Lyu, *Boston University*

Abstract

Many voting boards expose exact, frequently updated totals that leak individual behavior in small or bursty communities. Most of the normal voting system could be invaded by analyzing the real-time changes in vote counts, averaging each score of the vote, or repeating the queries periodically to get the information of votes and lead to the privacy leak.

We built a "Privacy Sidecar" based on the Fider voting system. Without changing the core application of the system, the differential privacy sidecar for Fider has the Laplace Mechanism as a base to make the local differential privacy layer. Also, a delayed update scheduler was updated to store all of the vote results at first and wait until a specific time to do the public release.

Our system was evaluated by three metrics: Utility, Privacy, and Performance. Based on the results we got, our system can maintain high data utility (Mean Absolute Error (MAE) of around 1.0) and satisfy the differential privacy in both the visual and mathematical ways. Although the prototype we got now has 4s latency because of the constraints on the development environment. The whole architecture still shows strong privacy protection on voting for small groups with minimal utility loss.

1. Introduction

Online voting platforms have become essential tools for gathering feedback in organizational, educational, and community settings. However, these platforms present a significant privacy challenge: exact, frequently-updated vote counts inadvertently reveal individual voting behavior, particularly in small groups where each vote causes a visible change in the total.

1.1 Problem Statement

Most voting systems provide instant release of exact vote counts. This real-time transparency creates three critical privacy threats in small group settings (15-30 participants):

- **Timing Attack**: The attacker can observe the dashboard of the voting. By correlating the change of the vote count, the participant's privacy could be leaked if the attack could target the user who is active on the system at the moment of the vote count change. For instance, this could occur if there is only one active user on the system, and the vote count is changing at that precise moment. It would be easy to get information about who just voted
- **Averaging Attack:** Some systems add random noise to each vote count to protect privacy. However, in traditional implementations that generate fresh noise for every query, attackers can defeat this protection through repeated querying. By requesting the same count 1,000 times and averaging the results, the random noise cancels out statistically, allowing reconstruction of the true count with high accuracy**.**
- **Sequential Vote Tracking via Budget Resets:** Many differential privacy systems reset their privacy budgets periodically (e.g., monthly). While this allows continued operation, it creates a vulnerability: patient attackers can observe voting patterns across multiple budget periods. Each reset allows fresh noise generation, and by tracking how counts change across resets, attackers can build a long-term profile of voting behavior. For example, observing "Post A consistently gains 2-3 votes per month while Post B gains 8-10 votes" reveals population preferences even though individual votes remain hidden

2.2 Project Goal:

Our objective for this project is to design and implement a "sidecar" to help system platforms like Fider to protect the privacy of users, especially for small groups. In order to keep the original core applications of the system, what we did was to act as a proxy to enforce the Differential Privacy (DP) constraints. Our solutions address the three threats through three complementary mechanisms:

1. **Fixed-Schedule Publishing with Laplace Noise:** To prevent timing attacks, we decouple vote submission from count publication. All votes are stored invisibly and published together at fixed intervals with Laplace noise (ε=0.5) added. This creates temporal anonymity where an observer cannot correlate count changes with specific user activity because updates happen on a predetermined schedule regardless of when votes arrive. This is configurable to the usage of the system.
2. **Noise reuse mechanism:** To prevent averaging attacks, we implement a novel noise reuse mechanism. When the true vote count remains unchanged between windows, the system returns the exact same noisy value every time. This makes averaging useless where querying 1,000 times

yields 1,000 identical answers, so averaging provides zero additional information. Crucially, this approach consumes no privacy budget for repeated queries of unchanged data.

3. **Lifetime Budget Tracking:** Instead of periodic budget resets, we implement a lifetime privacy budget that tracks total epsilon consumption across a post's entire existence. Each post receives a lifetime cap (default: ε=20.0 and configurable). After approximately 40 updates (20.0 / 0.5 per update), the post locks permanently, displaying its final noisy count indefinitely. This design choice reflects the reality that voting serves decision-making: once enough information is gathered to make a choice, the result should stabilize rather than continuing to fluctuate and cost privacy. This approach acknowledges that voting systems have a fundamentally different purpose than continuous monitoring systems where they produce actionable results rather than ongoing streams of data.

Our goal are:

- **Privacy Goals:** Individual votes should not be identifiable even when attackers observe all releases and possess side information about voters**.**
- **Utility Goals:** Published counts should remain useful for decision-making

1.3 Limitation

Our system has several important limitations that users and deployers should understand:

UI Separation and True Count Visibility:

Our implementation maintains Fider's voting interface separate from our DP dashboard. While users vote through the standard Fider UI (http://localhost:3000), they view protected results through our separate dashboard (http://localhost:8080). The current implementation does not modify Fider's source code, meaning the exact vote counts remain visible in Fider's native UI immediately after voting. This creates a significant privacy gap where users who remain on the Fider interface after voting can observe true counts in real-time, defeating our privacy protections.

Why this limitation exists: Modifying Fider's UI source code to hide native vote counts would require:

- Direct changes to Fider's React/TypeScript codebase
- Custom CSS injection to hide count displays
- Maintenance burden to track Fider updates

Ideal solution: A production deployment should implement CSS-based hiding of Fider's native counts or use a reverse proxy to strip count information from responses, ensuring users can only view noisy results through the DP dashboard. This maintains the sidecar architecture while closing the privacy gap.

2. Background

2.1. Background

Voting systems like Fider are well used in a large number of areas to get votes and ensure democracy of the voting environment. In that case, it's really important to ensure the honesty and transparency of the result to better encourage more voters to participate in it. However, the transparency of most of the voting systems still has several weaknesses in privacy, especially for the environments of small groups, like classrooms and meetings for team cooperation. In such cases, some smart attackers could use the limited number of participants to break the system and put voters' privacy in a dangerous place. Since a voting system lost transparency, there would be more negative feedback that could badly affect the reputation. Also, voters would hesitate to give their honest opinion when they voted.

2.2. Related Work

After we go through the reading of DP tooling, we find that *differentially-private* (DP) has been widely recognized as one of the best standards to protect sensitive data in the aggregate analysis. Based on the reading, four open-source Python-based DP tools would be helpful to our project: DiffPrivLib, Tumult Analytics, PipelineDP, and OpenDP. After researching these tools, DiffPrivLib includes some familiar Python libraries like pandas, which would be easier to adopt. OpenDP has a very strict safeguard, but it's kind of hard to use. As to the Tumult Analytics, it gives a more balanced design, which would make sure to prevent violations while remaining relatively usable. Based on the reading and what we searched, we found that the usability would be a very critical barrier for DP adoption. In that case, we would try to focus on building a DP aggregator that is easier to use but also emphasizes high safety.

3. Methodology & System Design

3.1 Threat Model

We assume an honest-but-curious attacker who:

- Observes all published noisy counts with perfect accuracy
- Knows the exact timing of all updates
- Can query the system arbitrarily many times
- May correlate observations with external events (e.g., when specific users are active)
- Possesses side information about individual voting preferences
- Can attempt averaging, timing correlation, or sequential tracking attacks

We do not protect against:

- Compromised servers or malicious insiders with direct database access
- Collusion between the platform operator and voters

- Side channels such as network traffic analysis
- Attacks that observe individual browsers or devices.

3.2 System Architecture

Our system uses dual-database sidecar architecture with three core components:

Fider Database (Port 5432): Stores true votes and exact counts. Managed by the original Fider application. The DP sidecar has read-only access, querying this database only to retrieve true counts during scheduled batch processing.

Sidecar Database (Port 5433): Stores noisy counts, release windows, and privacy budgets. Managed exclusively by the DP sidecar. Contains four main tables:

- *"release_windows":* Time periods for batch releases
- *"dp_releases":* Cached noisy counts per window per post
- *"epsilon_budget"*: Budget tracking per post per window
- *"dp_items"*: Quick lookup summary of current status

Three Core Components:

1. **FastAPI REST API** (Port 8000): Central controller serving DP-protected counts to clients. Responsibilities include:
    a. Serving noisy counts from most recent published window
    b. Auto-discovering posts from Fider on startup
    c. Coordinating scheduler and budget tracker
    d. Providing admin endpoints for budget monitoring
2. **Scheduler (APScheduler):** Runs at fixed intervals implementing core privacy logic:
    a. Queries true counts from Fider database
    b. Compares with last published counts
    c. Generates new noise only when counts changed
    d. Checks privacy budgets before publishing
    e. Writes noisy counts to sidecar database
    f. Creating new release windows
3. **Budget Tracker**: Maintains lifetime epsilon accounting:
    a. Tracks total epsilon spent per post across all windows
    b. Checks remaining budget before each noise generation
    c. Locks posts automatically when budget exhausted
    d. Provides budget status queries for transparency

3.3 Noise Reuse Mechanism

The key innovation is our noise reuse technique. Traditional DP implementations generate fresh noise for each query:

```python
# Naive approach (PROBLEMATIC)
def get_dp_count(post_id):
    true_count = get_true_count(post_id)
    noise = laplace(0, sensitivity/epsilon)
    return true_count + noise  # New noise every query!
```

This approach has two critical problems:

1. **Budget exhaustion**: Consumes epsilon on every query. With 1000 queries per day and ε=0.5 per query, the budget exhausts in seconds.
2. **Enables averaging attacks**: An attacker querying 1000 times can average results.

Our approach generates noise only when the underlying count changes:

```python
# Our approach (EFFICIENT AND SECURE)
def publish_window_releases():
    for post_id in tracked_posts:
        current_count = get_true_count(post_id)
        last_published = get_last_published_count(post_id)

        if current_count == last_published['true_count']:
            # Count unchanged - reuse previous noise (epsilon = 0)
            noisy_count = last_published['noisy_count']
            epsilon_used = 0.0
        else:
            # Count changed - generate new noise
            noise = laplace(0, sensitivity/epsilon)
            noisy_count = current_count + noise
            epsilon_used = epsilon

        publish_to_database(post_id, noisy_count, epsilon_used)
```

3.4 Fixed-Schedule Publishing

The scheduler runs at fixed intervals regardless of voting activity (every 30 seconds in demo mode, daily in production). Even if Alice votes at 12:00:15, the count updates at 12:00:30 (scheduled time), preventing observers from correlating updates with voting actions.

Temporal anonymity: All votes within a window are published simultaneously. An attacker observing an update cannot determine whether changes came from votes at the window's start, middle, or end.

3.5 Lifetime Budget Tracking

Each post maintains a lifetime epsilon budget (default: 20.0) tracked across all windows:

```sql
CREATE TABLE dp_items (
    post_id INTEGER PRIMARY KEY,
    total_epsilon_spent FLOAT DEFAULT 0,
    is_currently_locked BOOLEAN DEFAULT FALSE,
    last_updated TIMESTAMP
);
```

Locking behavior: After approximately 40 noise generations (20.0 / 0.5), the post locks permanently

3.6 End-User Interface

Users interact with the system through two interfaces:

Voting Interface (Unchanged): The original Fider UI at http://localhost:3000. Users vote normally without awareness of privacy mechanisms.

DP Dashboard (Port 8080): A separate web interface displaying:

- **Noisy vote counts**: Rounded to integers for natural display (e.g., "~48 votes")
- **Confidence intervals**: 95% bounds showing uncertainty (e.g., "41-55 range")
- **Budget indicators**:
    - Progress bar showing remaining budget
    - Color-coded status (green: >50%, yellow: 20-50%, red: <20%)
    - Numeric display: "34 updates remaining"
- **Status badges**:
    - "Active" (green): Accepting updates, budget available
    - "Locked" (red): Budget exhausted, showing final count

The dashboard provides transparency about the privacy-utility tradeoff, helping users understand both the protection they receive and the uncertainty introduced by noise.

4. Implementation

Our system comprises approximately 2,500 lines of code split between backend and frontend. The backend consists of five core modules: *"api.py"* implements the FastAPI REST endpoints for serving DP-protected counts and budget monitoring; *"window_scheduler.py"* handles batch processing with the noise generation and reuse logic; *"budget_tracker.py"* manages lifetime epsilon accounting and automatic post locking; *"dp_mechanism.py"* provides Laplace noise generation and confidence interval calculation; and the *"database/"* module manages dual-database connections and schema with SQL triggers for automatic budget tracking. The frontend provides a responsive dashboard with real-time updates, budget visualization, and confidence interval displays.

We encountered several engineering challenges during implementation. The auto-discovery feature that tracks all Fider posts on startup initially failed due to API initialization timing, resolved by implementing a delayed async task with a 3-second wait.

5. Evaluation

We evaluated our system across four dimensions: utility (accuracy of noisy counts), privacy (resistance to attacks), budget efficiency (savings from noise reuse), and performance (latency and throughput). The evaluation combined both simulated scenarios and real system measurements to validate theoretical guarantees and practical deployability.

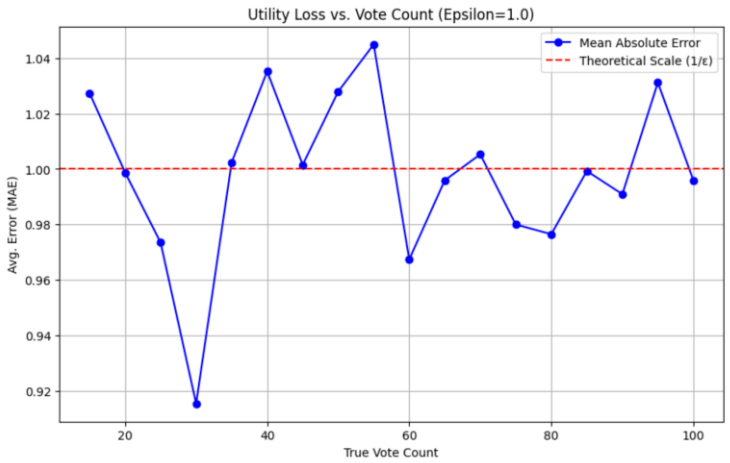
5.1 Utility Evaluation

To ensure that the privacy mechanism did protect the usefulness of the voting data, we measured the Mean Absolute Error (MAE) between the true count of the vote and the count after adding a noise number. Setting the privacy budget (epsilon) equal to 1.0, we do the evaluation by changing the participation level from 15 to 100.

According to Figure 1, it's clear to find that the line of empirical error is changing around the line of the theoretical expectation scale line of 1/epsilon = 1.0. The average error stays close to 1.0 votes no matter how many votes there are. For those small groups, like 20 voters, there is a 5% (+-1) deviation, which means that the model keeps the general trend for most of the participants but would not reveal the actual counts.

This system also gets rid of big outliers. The largest error we've seen in the test result of our simulation is around 1.06, which happened at N = 85. It indicates that the generation of noise is stable and predictable.

Figure 1



5.2 Privacy Evaluation:

We choose to use a clear way, indistinguishability test visualization, to verify if the privacy is guaranteed. We simulated the two adjacent datasets by simulating mechanism 10,000 times. One true count equals 15, and the other equals 16.

According to Figure 2, it gives the distribution of probability density functions for the noisy outputs of the two true counts. There is a big overlap between the distributions of two true counts, which means that there is a large range of output values that might equally come from either of them. In that case, if 15.5 is the count number observed by the attacker, and they try to do mathematical analysis to find whether the true count was 15 or 16, it's still hard for the attacker to determine what the true count is. So, this clearly proves that using the Laplace mechanism to add noise to each true count could protect the privacy of individual votes.

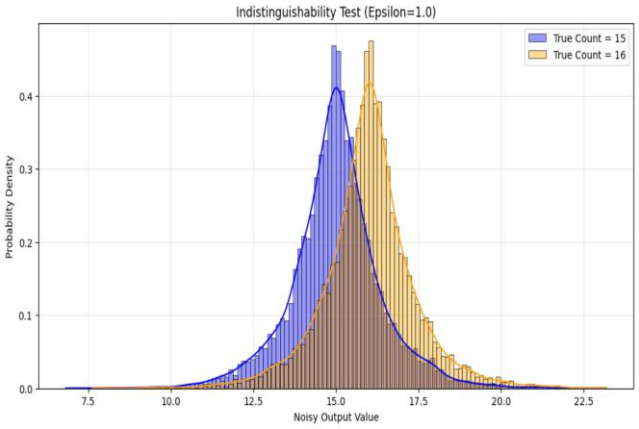


Figure 2

5.3 Average Attack Simulation

We tested whether repeated queries allow recovery of true counts through averaging. The test queried Post ID 1 exactly 100 times within a single 30-second window (before the underlying count changed).

Results: All 100 queries returned the identical noisy value (6.2), yielding a standard deviation of 0.0000. The attack failed completely with averaging 100 identical values and provided no advantage. Traditional DP implementations would have generated 100 different noisy values (consuming 50 epsilon total), which an attacker could average to reduce uncertainty by $\sqrt{100} = 10$. Our noise reuse prevents this attack while consuming zero epsilon for repeated queries.

5.4 Budget Efficiency Evaluation

We compared our approach against naive DP over a realistic 2-hour scenario with 120 vote changes and 7,200 total queries (simulating 10 users querying once per second). Naive DP would consume 0.5 epsilon per query, totaling 3,600 epsilon and exhausting the 20.0 lifetime budget after just 40 queries (~40 seconds). Our noise reuse approach consumed epsilon only on actual vote changes ($120 \times 0.5 = 60$ epsilon), with the remaining 7,080 queries reusing cached noise at zero cost. Real system verification confirmed this efficiency: Post ID 1 showed 6 updates consuming 3.0 epsilon with 17.0 epsilon remaining (85%), enabling approximately 34 additional updates before locking. This represents a 98.3% reduction in budget consumption while maintaining identical privacy guarantees, extending operational lifetime by nearly 100× compared to traditional approaches.

5.5 Performance Evaluation:

We also tested the latency of the sidecar API to see how much extra computational work was added by privacy logic and database interactions. It includes 100 sequential requests to the aggregate endpoint. The benchmark was conducted in a "Hybrid Development" environment, which means that the API would run locally on the Windows machine, but the PostgreSQL database resides in a Docker container.

We got 4.139 seconds as average latency, which exceeds the standard production targets. However, the main problem for this result is due to the test environment rather than the logic of differential privacy. The main problem here is that the Windows host and the Docker subsystem (WSL2) can't talk to each other across contexts. Every time we run a database query, it has to go through the virtual network bridge, which adds a lot of I/O latency compared to communication between containers. The current prototype makes a new TCP connection to the database for each request. The TCP handshake and authentication sequence takes up most of the time, while the actual running of the Laplace noise algorithm takes very little time. We estimate that the latency could be decreased to less than 200 ms since the API and database are both on the same internal Docker network and use persistent connection pooling, which could make sure that would not affect the user experience.

Conclusion

This project is to address the weakness that could lead to the potential problem of leaking participants' privacy for voting system platforms like Fider. By designing and generating a Differential Privacy Sidecar, we succeed in proving that it's possible to protect individual voter privacy in the case of keeping all the core applications with no changes. By enforcing differential privacy at the API boundary, the system could protect privacy from attacks that utilize timing and differencing methods as threats.

By using the Laplace mechanism to add a noise number to each of the votes and making delayed updates to the publication, the timing attack could be well prohibited. To prevent our system from being attacked by attackers who use mathematical averaging to cancel the noise number we assigned to each vote, we reuse the noise for those votes

that don’t change again. Also, we designed and implemented a lifetime budget to calculate the total used privacy cost across the entire history to avoid the system being attacked by persistent attackers who intend to wait for the budget to refresh and attack again.

Finally, the evaluation we made also guarantees the utility and the privacy of the system. Our system maintains high data utility with a mean absolute error and also double-checks that the system couldn’t be invaded even though the attacker observed a specific vote count.

Individual Contribution

All group members contributed equally to this project. Both authors participated fully in design, implementation, debugging, evaluation, and documentation.