Batch Correlation in Mixnets: Computing Mapping Probabilities

Aiswarya Walter, Student ID: 427199

1 Problem Definition

Mix networks (mixnets) are anonymous communication systems that provide privacy protection by preventing adversaries from tracing communication patterns and linking senders to recipients. Unlike low-latency systems such as Tor, mixnets introduce deliberate delays and message mixing strategies to ensure unlinkability even against powerful adversaries capable of global network monitoring. They operate by routing messages through a series of intermediary nodes known as mixes, which shuffle and delay packets to disrupt any correlation between input and output messages, making it difficult for observers to correlate incoming and outgoing traffic patterns.

One critical challenge in mixnet design is handling node failures, which can compromise message delivery and system reliability. One of the strategies to address this challenge is message splitting, where a single message is divided into multiple sub-messages that are sent as a batch. The original message can be reconstructed when a sufficient number of sub-messages in the batch are successfully received at the destination, typically using erasure coding or redundancy schemes. This approach provides fault tolerance by allowing message recovery even when some sub-messages are lost, while maintaining the mixing properties of the network.

The challenge arises when a global adversary observes the mixnet system. Such an adversary can monitor all incoming and outgoing traffic at the mix nodes, recording message arrival and departure timestamps, batch compositions, and routing patterns. This batch-level correlation problem is particularly challenging because temporal patterns, batch sizes, and network timing constraints can leak information about the mapping between input and output batches. Even though individual messages within batches may be cryptographically protected, the observable metadata (timing, message counts) can be exploited to reduce the anonymity provided by the mixnet.

Problem Statement: Given a mixnet system operating with message batches under observation by a global adversary, the aim is to develop an algorithm that computes the probability that each observed outgoing batch corresponds to each possible incoming batch, considering timing constraints and batch coherence requirements.

1.1 Assumptions

The proposed batch matching algorithm in Section 2 operates under several assumptions:

- The model assumes no message loss during transmission, reliable mix nodes without failures, crashes, or Byzantine behavior. In practice, networks experience failures, congestion, and malicious behavior that would impact batch reconstruction and correlation analysis.
- The model assumes fixed batch sizes and predictable processing patterns. Real mixnets may employ variable batch sizes, padding, dummy messages, and complex batching strategies that could alter correlation patterns and anonymity properties.
- The model does not include countermeasures such as traffic padding, cover traffic, dummy messages, or other defensive techniques that real mixnets might deploy to enhance anonymity against traffic analysis attacks.
- The adversary is assumed to have perfect global observation with complete visibility into all incoming and outgoing traffic and precise timestamps. This may not reflect realistic monitoring constraints or partial network visibility.
- The evaluation focuses on a single Poisson mix node rather than more complex multi-hop mixnet topologies that could provide additional anonymity guarantees.

1.2 Anonymity Metrics

The following anonymity metrics are used to quantify the privacy provided by mixnet systems:

- **Batch Mapping Probabilities:** The probability that each outgoing batch originated from each possible incoming batch, representing the uncertainty an adversary faces when attempting to correlate batches.

- **Anonymity Set:** The collection of incoming batches that could plausibly have contributed to a given outgoing batch. Larger anonymity sets indicate stronger privacy protection.
- **Anonymity Set Size:** The number of possible incoming batches for each outgoing batch, which is the cardinality of the anonymity set of the given outgoing batch.

2 Proposed Solution: The Mixnet Batch Matching Algorithm

Variables and Functions	**Description**	**Data Type**
M_{ij}	j-th message in the incoming batch i	Message object
O_{pq}	q-th message in the outgoing batch p	Message object
$t_{M_{ij}}$	Arrival time of message M_{ij}	Timestamp
$t_{O_{pq}}$	Sending time of message O_{pq}	Timestamp
$IncomingBatches$	Mapping of incoming batch ids to its messages	Dictionary of dictionaries
$OutgoingBatches$	Mapping of outgoing batch ids to its messages	Dictionary of dictionaries
$OutBatchMappingCount$	For each outgoing batch, it maps the candidate incoming batches to the number of valid permutations to that batch	Dictionary of dictionaries
$OutMsgMappingSet$	Set of valid input messages for each outgoing message, O_{pq}	Dictionary of sets of messages
$Valids$	List of valid message permutations for all outgoing batches	List of dictionaries of lists
$BatchProb$	Probability mapping for all output batches	Dictionary of dictionaries
$AnonymitySet$	Set of incoming batches that contribute to the anonymity of each outgoing batch	Dictionary of sets of messages
$AnonymitySetSize$	Number of incoming batches that could have contributed to each outgoing batch	Dictionary of integers
$batchid(msg)$	Returns the batch id of the message	Function (returns integer)
$msgid(msg)$	Returns the message id	Function (returns integer)
$appendMsg(subpermutation, msg)$	Operation to append a message to an existing sup-permutation of a batch	Function (returns list)

Table 1: Algorithm variables and functions.

The Mixnet Batch Matching algorithm addresses the batch correlation problem by systematically enumerating all valid message-to-batch mappings and computing probability distributions for each outgoing batch. The algorithm operates incrementally, processing each outgoing message as it enters a batch and updating the global set of valid permutations (see $Valids$ in Table 1).

One of the algorithm's main data structures is a batch mapping **permutation** (elements of $Valids$), which represents a complete assignment of incoming messages to outgoing batch positions. Specifically, each permutation is a mapping from outgoing batch IDs to lists of incoming messages, where each list represents the possible source incoming messages that could have contributed to that outgoing batch.

For example, consider a scenario with three outgoing batches (A, B, C) and incoming messages from batches 1, 2, and 3. A valid permutation might be:

$$\begin{aligned} \text{Permutation} = \{\text{A} : [M_{11}, M_{12}], && (1)\\ \text{B} : [M_{21}], && (2)\\ \text{C} : [M_{31}, M_{32}, M_{33}]\} && (3) \end{aligned}$$

This permutation hypothesizes that outgoing batch A originated from two messages in incoming batch 1, outgoing batch B came from one message in incoming batch 2, and outgoing batch C derived from three messages in incoming batch 3.

Each individual mapping within a permutation (e.g., A: $[M_{11}, M_{12}]$) is called a **sub-permutation**, representing the specific assignment of incoming messages to a single outgoing batch position. Each outgoing batch has its own sub-permutation defining which incoming messages could have contributed to it.

The algorithm maintains two constraints for valid permutations:

- All messages mapped to the same outgoing batch must originate from the same incoming batch.
- Only incoming messages that arrived before an outgoing message was sent can be candidates for mapping to that outgoing batch, ensuring realistic timing relationships.

The algorithm explores all permutations that satisfy these constraints, counting how many valid mappings exist for each outgoing-to-incoming batch pair. The final probability that outgoing batch p originated from incoming batch i is computed as:

$$P(\text{outgoing batch } p \leftarrow \text{incoming batch } i) = \frac{\text{number of valid permutations mapping } p \text{ to } i}{\text{total number of valid permutations}}$$

Table 1 provides a comprehensive overview of the key variables, data structures, and functions used throughout the algorithm. The algorithm maintains several core data structures: `Valids` stores the current set of valid permutations, `OutBatchMappingCount` tracks the number of valid mappings for each outgoing-incoming batch pair, `BatchProb` contains the final probability distributions, and `OutMsgMappingSet` holds the eligible incoming messages for each outgoing batch. The algorithm also maintains anonymity-related structures: `AnonymitySet` stores the set of incoming batches that could have contributed to each outgoing batch, while `AnonymitySetSize` records the cardinality of these anonymity sets for privacy quantification. The helper functions `batchid()` and `msgid()` extract batch and message identifiers respectively, while `appendMsg()` handles the creation of extended sub-permutations during the permutation expansion phase.

The following subsections describe each phase of the algorithm in detail, providing both pseudocode and comprehensive explanations of the underlying logic.

2.1 Initialization and Candidate Selection

Phase 1 sets up the fundamental data structures and identifies eligible messages for batch mapping. In lines 1-4, the algorithm initializes empty containers for the core data structures: `Valids` for storing valid permutations, `BatchProb` for probability distributions, and `AnonymitySet` and `AnonymitySetSize` for anonymity metrics. Line 5 records the arrival of outgoing message O_{pq} in batch p at timestamp $t_{O_{pq}}$, and line 6 initializes the `OutMsgMappingSet[`O_{pq}`]` as an empty set to store eligible incoming messages for the current outgoing message.

Lines 7-13 implement the candidate selection logic by iterating through all incoming batches and applying size-based filtering. For each incoming batch i, the algorithm compares its size ($lenIn$) with the current outgoing batch size ($lenOut$) in line 10. Only incoming batches that satisfy the constraint $lenIn \geq lenOut$ are considered as candidates (line 11). This creates the initial structure `OutBatchMappingCount[p][i]` with a count of zero for each eligible incoming batch.

The temporal filtering occurs in lines 14-20, where the algorithm examines each message in the candidate incoming batches. Lines 14-15 establish nested loops that iterate through each eligible incoming batch i in `OutBatchMappingCount[p]` and each message M_{ij} with timestamp $t_{M_{ij}}$ in that batch. For every such message, the algorithm applies the causality constraint $t_{M_{ij}} < t_{O_{pq}}$ (line 16). Only messages that arrived before the current outgoing message was sent are added to `OutMsgMappingSet[`O_{pq}`]` (line 17), ensuring temporal consistency in the batch mapping.

Phase 1 Initialization and Candidate Selection

```
1:  $Valids \leftarrow []$
2:  $BatchProb \leftarrow \{\}$
3:  $AnonymitySet \leftarrow \{\}$
4:  $AnonymitySetSize \leftarrow \{\}$
5:  Outgoing message $O_{pq}$ enters the outgoing batch, $p$ at time $t_{O_{pq}}$
6:  $OutMsgMappingSet[O_{pq}] \leftarrow \{\}$
7:  for each $i$ in $IncomingBatches$ do
8:      $lenIn \leftarrow len(IncomingBatches[i])$
9:      $lenOut \leftarrow len(OutgoingBatches[p])$
10:     if $lenIn \geq lenOut$ then
11:         $OutBatchMappingCount[p][i] \leftarrow 0$
12:     end if
13: end for
14: for each $i$ in $OutBatchMappingCount[p]$ do
15:     for each $M_{ij}, t_{M_{ij}}$ in $IncomingBatches[i]$ do
16:         if $t_{M_{ij}} < t_{O_{pq}}$ then
17:             $OutMsgMappingSet[O_{pq}].add(M_{ij})$
18:         end if
19:     end for
20: end for
```

2.2 Initial Permutation Construction

Phase 2 handles the base case when no valid permutations have been established yet. Line 1 checks if the `Valids` list is empty, indicating this is the first outgoing message being processed. In this scenario, lines 2-4 iterate through each eligible incoming message M_{ij} from the $OutMsgMappingSet[O_{pq}]$ collection established in Phase 1.

For each eligible message, line 3 creates a new permutation represented as a dictionary where the key p (the current outgoing batch ID) maps to a list containing the single message M_{ij}. This creates the foundational permutation structure $\{p : [M_{ij}]\}$ that will be expanded in subsequent phases. Each such permutation is appended to the `Valids` list, establishing the initial set of valid mappings that serve as the basis for all future extensions.

Phase 2 Initial Permutation Construction

```
1: if $Valids = []$ then
2:     for each $M_{ij} \in OutMsgMappingSet[O_{pq}]$ do
3:         $Valids \leftarrow Valids.append(\{p : [M_{ij}]\})$
4:     end for
5: end if
```

2.3 Permutation Extension and Validation

Phase 3 implements the core expansion logic when valid permutations already exist. Line 1 checks that `Valids` is not empty, confirming that initial permutations have been established. Line 2 initializes `tempValids` as a temporary container for the new set of extended permutations.

Lines 3-5 begin the message processing loop, where each eligible incoming message M_{ij} is extracted along with its batch ID (i) and message ID (j) using the helper functions `batchid()` and `msgid()`. Lines 7-9 initialize the permutation extension variables: `newX` as a copy of the current permutation x, `count` as a validation counter, and `msgList` as the existing sub-permutation for outgoing batch p.

The algorithm handles two distinct cases based on whether a sub-permutation already exists for batch p. In Case 1 (lines 10-29), when `msgList` exists, lines 12-20 implement the coherence validation logic. The inner loop examines each existing message in the sub-permutation, extracting its batch ID (b) and message ID (m) in lines 13-14. The coherence check occurs in line 15: if all existing messages come from the same incoming batch ($b = i$) and the new message is distinct ($m \neq j$), the count is incremented; otherwise, the validation fails and the loop breaks.

If the coherence check passes (line 21), indicating all messages in the sub-permutation plus the new message belong to the same incoming batch, lines 22-26 extend the sub-permutation by appending M_{ij} using `appendMsg()`, updating the permutation structure, adding it to `tempValids`, and incrementing the mapping count for batch pair (p, i).

Case 2 (lines 30-45) handles the creation of new sub-permutations when no mapping exists for batch p. Lines 32-36 implement the uniqueness constraint by checking that incoming batch i has not been used elsewhere in the current permutation. If batch i is unique across the entire permutation (line 37), lines 38-40 create a new sub-permutation containing only M_{ij}, add the extended permutation to `tempValids`, and update the mapping count.

Lines 48-49 finalize the phase by replacing the old `Valids` with the newly constructed `tempValids` and resetting the temporary container for the next iteration.

2.4 Global Batch Mapping Update

Phase 4 ensures consistency across all outgoing batch mappings after the current batch has been processed. Lines 1-2 establish nested loops that iterate through every valid permutation x in `Valids` and every outgoing batch ID (`outid`) within each permutation.

Line 3 implements a filtering condition to skip the current batch p, focusing only on previously processed outgoing batches that need count updates. For qualifying batches, line 4 extracts the incoming batch ID (`inId`) from the first message in the sub-permutation, leveraging the batch coherence property that ensures all messages in a sub-permutation originate from the same incoming batch.

Line 5 performs the count increment operation by updating `OutBatchMappingCount[outid][inId]`, ensuring that the mapping counts for all outgoing batches remain synchronized with the current set of valid permutations. This maintains the integrity of probability calculations across the entire system.

Phase 3 Permutation Extension and Validation

```
if $Valids \neq []$ then
    $tempValids \leftarrow []$
    for each $M_{ij} \in OutMsgMappingSet[O_{pq}]$ do
        $i \leftarrow batchid(M_{ij})$
        $j \leftarrow msgid(M_{ij})$
        for each $x$ in$Valids$ do
            $newX \leftarrow x$
            $count \leftarrow 0$
            $msgList \leftarrow newX.get(p)$
            if $msgList$ exists then
                                                    ▷ Case 1: Extend existing sub-permutation
                for $v \leftarrow 0$ to $len(msgList)$ do
                    $b \leftarrow batchid(msgList[v])$
                    $m \leftarrow msgid(msgList[v])$
                    if $b = i$ and $m \neq j$ then
                        $count \leftarrow count + 1$
                    else
                        break
                    end if
                end for
                if $count = len(msgList)$ then
                    $newMsgList \leftarrow appendMsg(msgList, M_{ij})$
                    $newX[p] \leftarrow newMsgList$
                    $tempValids \leftarrow tempValids.append(newX)$
                    $OutBatchMappingCount[p][i] \leftarrow OutBatchMappingCount[p][i] + 1$
                    $count \leftarrow 0$
                else
                    $count \leftarrow 0$
                end if
            else
                                                    ▷ Case 2: Create new sub-permutation
                for each $batchMsgs$ in $x$ do
                    if batchid($batchMsgs[0]$) $\neq i$ then
                        $count \leftarrow count + 1$
                    end if
                end for
                if $count = len(x)$ then
                    $newX[p] \leftarrow [M_{ij}]$
                    $tempValids \leftarrow tempValids.append(newX)$
                    $OutBatchMappingCount[p][i] \leftarrow OutBatchMappingCount[p][i] + 1$
                    $count \leftarrow 0$
                else
                    $count \leftarrow 0$
                end if
            end if
        end for
    end for
    $Valids \leftarrow tempValids$
    $tempValids \leftarrow []$
end if
```

Phase 4 Global Batch Mapping Update

```
for each $x$ in $Valids$ do
    for each $outid$ in $x$ do
        if $outid \neq p$ then
            $inId \leftarrow \text{batchid}(x[outid][0])$
            $OutBatchMappingCount[outid][inId] \leftarrow OutBatchMappingCount[outid][inId] + 1$
        end if
    end for
end for
```

2.5 Probability Computation and Finalization

Phase 5 computes the final probability distributions and anonymity metrics. Lines 2-5 iterate through all outgoing batches in `OutBatchMappingCount`, initializing empty probability dictionaries for new batches and creating a `nonZero` container for valid mappings.

Lines 6-11 implement the probability calculation logic for each incoming-outgoing batch pair. Line 7 computes the probability as the fraction of valid permutations that map the incoming batch to the outgoing batch. Lines 8-10 filter out zero probabilities, storing only meaningful mappings in `nonZero`.

Lines 12-20 handle the finalization of results based on whether valid mappings exist. If `nonZero` contains valid mappings (line 12), lines 13-15 populate the final data structures: `BatchProb` receives the probability distribution, `AnonymitySet` stores the set of contributing incoming batches, and `AnonymitySetSize` records the cardinality of the `AnonymitySet`. If no valid mappings exist (lines 16-19), any existing entries for the outgoing batch are removed from `BatchProb`.

Lines 22-25 reset the global data structures to prepare for processing the next outgoing message, clearing `OutBatchMappingCount`, `BatchProb`, `AnonymitySet`, and `AnonymitySetSize` to ensure a clean state for subsequent iterations.

Phase 5 Probability Computation and Finalization

```
 1: for each $outBatch$ in $OutBatchMappingCount$ do
 2:     if $outBatch$ not in $BatchProb$ then
 3:         $BatchProb[outBatch] \leftarrow \{\}$
 4:     end if
 5:     $nonZero \leftarrow \{\}$
 6:     for each $(inBatch, count)$ in $OutBatchMappingCount[outBatch]$ do
 7:         $prob \leftarrow \frac{count}{len(Valids)}$
 8:         if $prob > 0$ then
 9:             $nonZero[inBatch] \leftarrow prob$
10:         end if
11:     end for
12:     if $nonZero$ then
13:         $BatchProb[outBatch] \leftarrow nonZero$
14:         $AnonymitySet[outBatch] \leftarrow \{nonZero.keys()\}$
15:         $AnonymitySetSize[outBatch] \leftarrow len(AnonymitySet[outBatch])$
16:     else
17:         if $outBatch$ in $BatchProb$ then
18:             $delBatchProb[outBatch]$
19:         end if
20:     end if
21: end for
22: $OutBatchMappingCount \leftarrow \{\}$
23: $BatchProb \leftarrow \{\}$
24: $AnonymitySet \leftarrow \{\}$
25: $AnonymitySetSize \leftarrow \{\}$
```

3 Evaluation

To assess the proposed Mixnet Batch Matching algorithm, we implemented it within a mixnet simulation framework called Mixim. The evaluation focuses on understanding how various system parameters affect anonymity metrics under the global adversary model.

3.1 Evaluation Metrics

The evaluation involves three anonymity metrics to assess the privacy guarantees provided by the mixnet system:

- **Number of Uniquely Identified Batches:** The count of outgoing batches that can be mapped to exactly one incoming batch with certainty (probability $= 1.0$). This metric represents the worst-case scenario where the adversary can definitively link input and output batches, completely breaking anonymity for those specific batches.
- **Average Anonymity Set Size:** The mean number of candidate incoming batches that could plausibly have contributed to each outgoing batch (i.e., batches with non-zero mapping probability). Larger anonymity sets indicate stronger privacy protection, as they represent the number of possible sources an adversary must consider when attempting to trace messages.
- **Accuracy Percentage:** The percentage of outgoing batches where the adversary's best guess (highest probability mapping) correctly identifies the actual source incoming batch. This metric quantifies the adversary's success rate in correlating batches, with lower percentages indicating better anonymity protection.

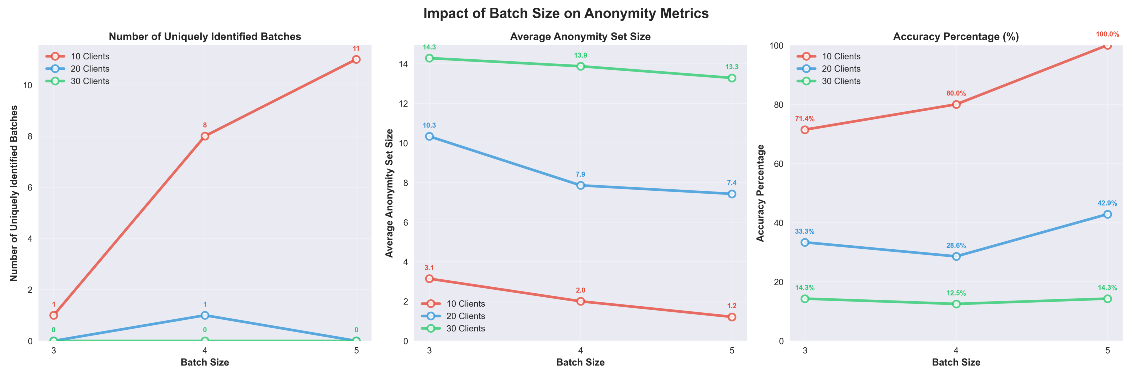

Figure 1: Impact of batch size on various anonymity metrics across different numbers of clients (10, 20, and 30 clients). The plots show how batch size affects the number of uniquely identified batches, average anonymity set size, and accuracy percentage.

3.2 Experimental Setup

Due to the computational complexity of enumerating all valid batch permutations and the resulting scalability limitations, we conducted a limited set of fixed runs for each parameter configuration. The experimental design covers the following parameter space:

- **Number of clients:** 10, 20, and 30 clients
- **Batch sizes:** 3, 4, and 5 messages per batch
- **Mix node configuration:** Single Poisson mix node
- **Simulation approach:** Fixed runs per configuration due to computational constraints

The evaluation metrics include the number of uniquely identified batches, average anonymity set size, and mapping accuracy percentage. These metrics provide insights into the anonymity guarantees offered by the mixnet under different parameters.

3.3 Observations

3.3.1 Impact of Batch Size on Anonymity Metrics

Figure 1 presents an analysis of how batch size affects anonymity across different client populations.

For systems with 10 clients (red line), increasing batch size from 3 to 5 results in a degradation of anonymity: the number of uniquely identified batches increases from 1 to 11, the average anonymity set size decreases from 3.1 to 1.2,

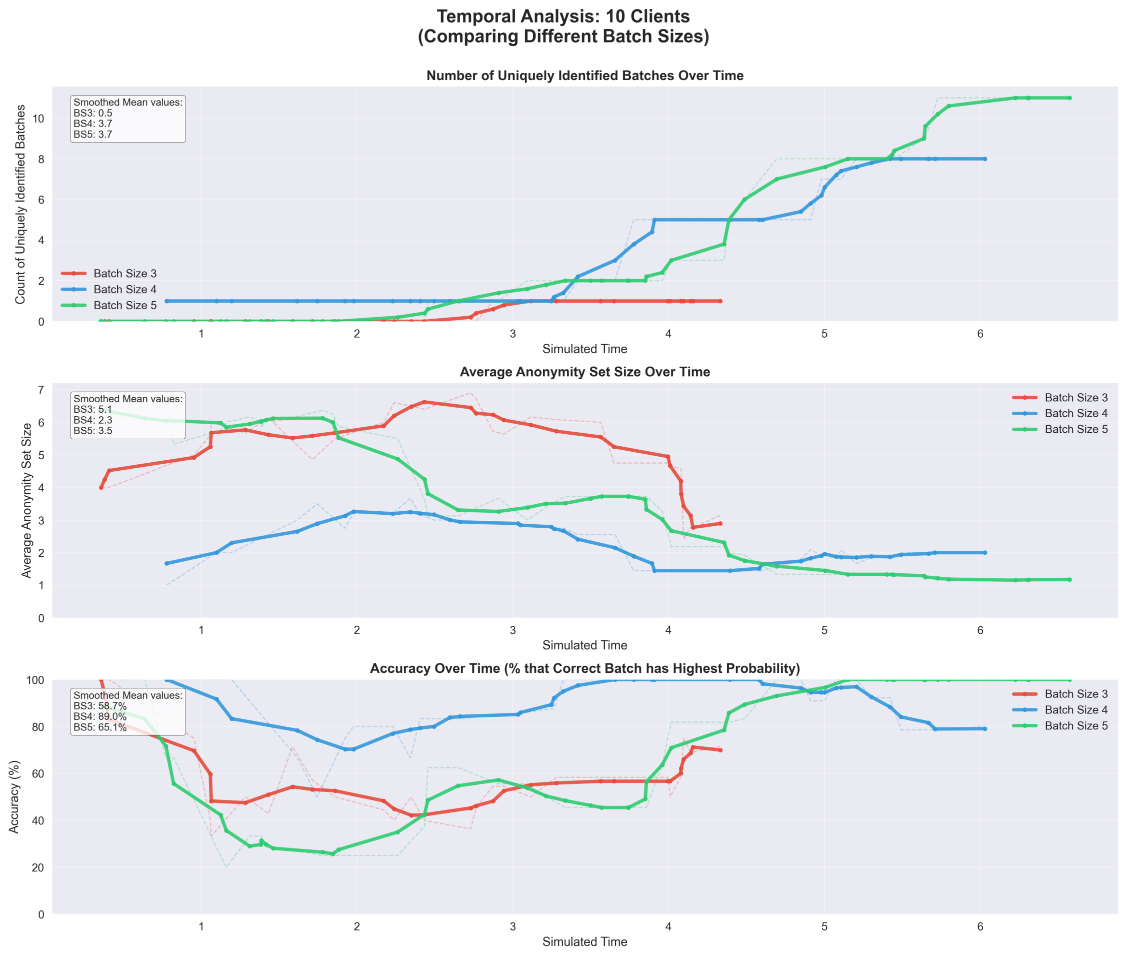

Figure 2: Temporal analysis of anonymity metrics for 10 clients across different batch sizes (3, 4, and 5).

and the accuracy percentage rises from 71.4% to 100%, indicating that larger batches make it easier for adversaries to correlate incoming and outgoing traffic.

In contrast, systems with 20 clients (blue line) show more complex behavior with some improvement in anonymity metrics at intermediate batch sizes, while systems with 30 clients (green line) maintain consistently strong anonymity across all batch sizes, with zero uniquely identified batches, stable anonymity set sizes around 13-14, and low accuracy percentages around 12-14%. This suggests that the impact of batch size on anonymity is dependent on the number of participating clients, where smaller client populations become increasingly vulnerable to traffic analysis as batch sizes increase, while larger client populations provide sufficient mixing entropy to maintain privacy regardless of batch size.

3.3.2 Temporal Analysis

The temporal analysis reveals how anonymity metrics evolve over time for different system configurations. Figures 2, 3, and 4 show the temporal patterns for 10, 20, and 30 clients respectively, with smoothed curves for different batch sizes.

The smoothing was applied to reduce noise and highlight underlying trends in the temporal data. A moving average filter with a window size of 5 data points was used to smooth the raw measurements.

For 10 clients (Figure 2), the temporal patterns show significant variation in anonymity metrics over time, with different batch sizes exhibiting distinct trajectories. The smaller client population appears to create more volatile anonymity guarantees. The 20-client configuration (Figure 3) demonstrates more stable temporal patterns, with the anonymity metrics showing smoother evolution over time. This suggests that moderate client populations may provide a good balance between mixing effectiveness and system stability. With 30 clients (Figure 4), the temporal analysis reveals the most consistent anonymity patterns across different batch sizes, indicating that larger client populations contribute to more predictable privacy guarantees.

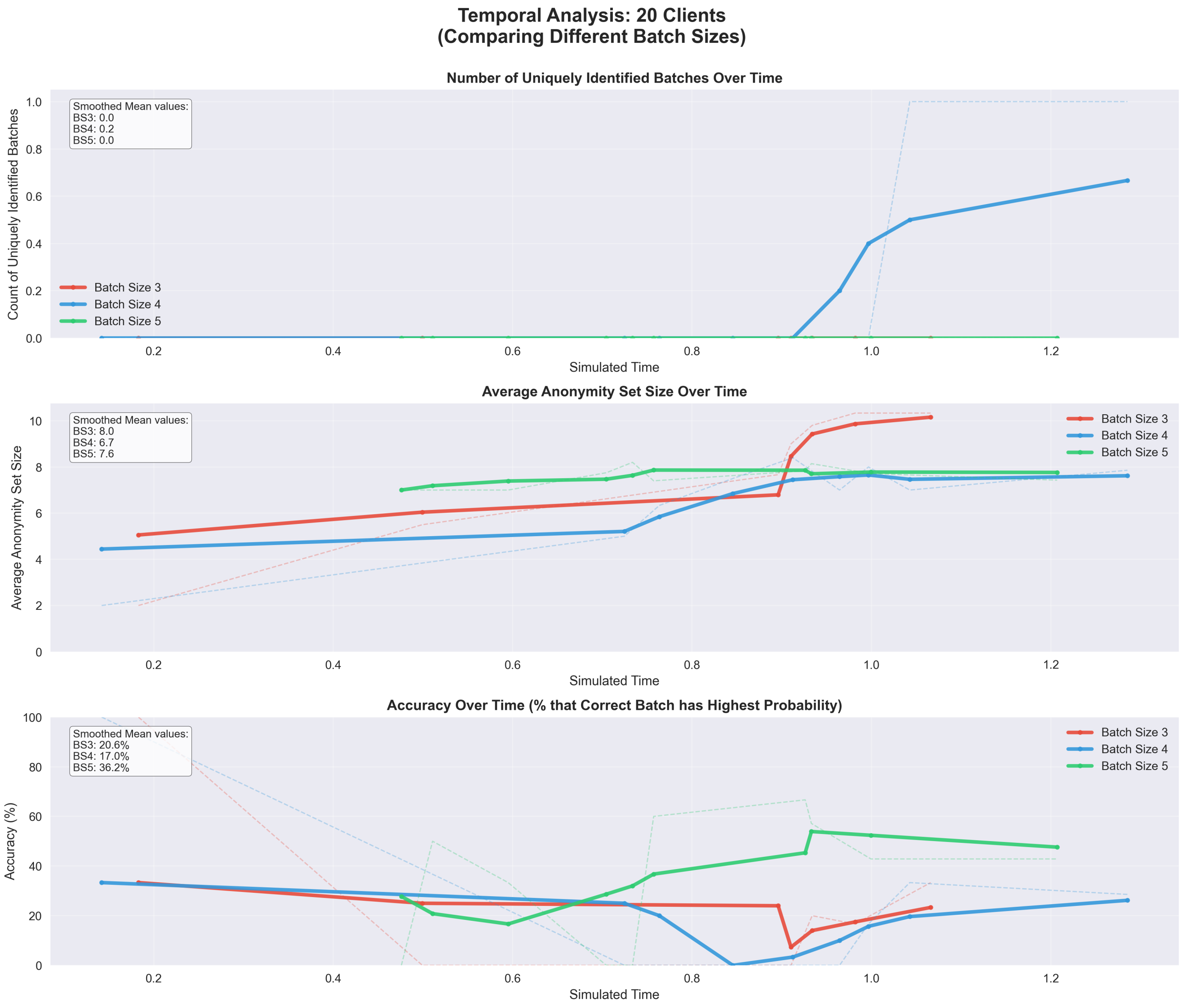

Figure 3: Temporal analysis of anonymity metrics for 20 clients across different batch sizes (3, 4, and 5).

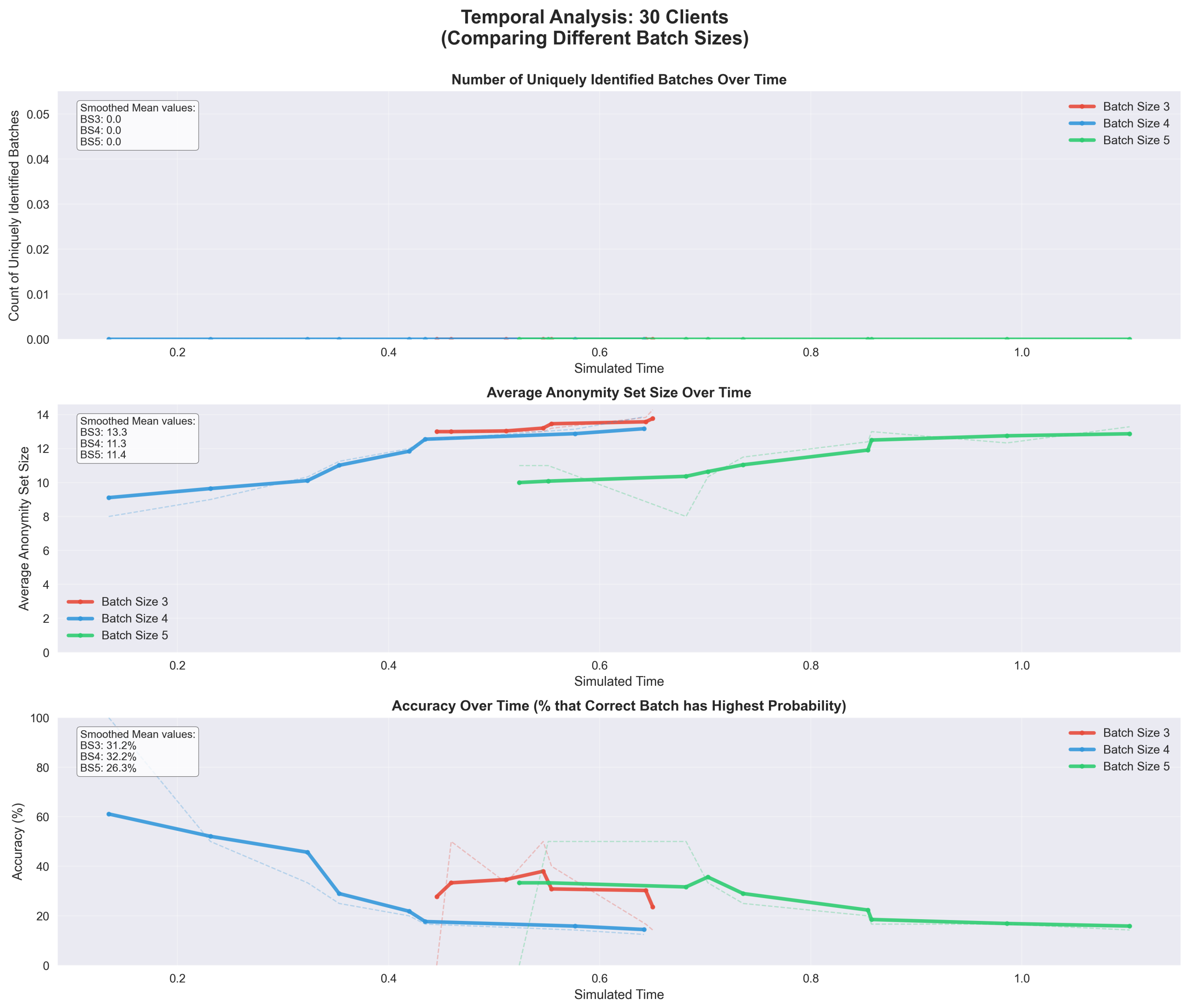

Figure 4: Temporal analysis of anonymity metrics for 30 clients across different batch sizes (3, 4, and 5).

3.3.3 Variation Between Runs

Figure 5 shows multiple runs of the same parameter configuration (20 clients, batch size 4) to assess the variability and consistency of results across independent simulation runs.

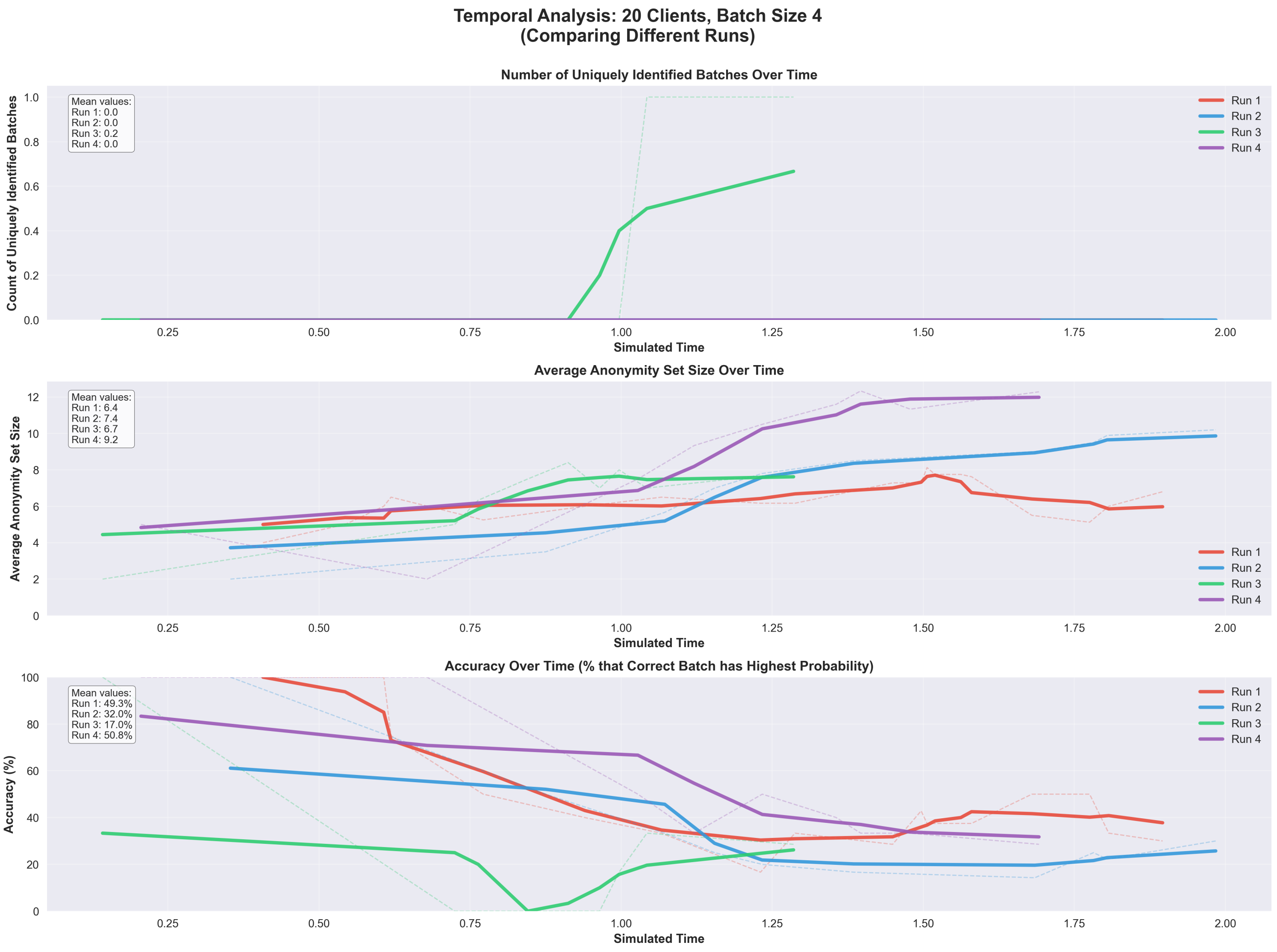

Figure 5: Temporal evolution of anonymity metrics across four independent simulation runs with identical parameters (20 clients, batch size 4).

The comparison reveals some variation between runs, highlighting the stochastic nature of the mixnet simulation and the importance of conducting multiple experiments for robust conclusions. The variation analysis across four independent runs reveals significant variability due to simulation randomness, with Run 3 showing notably different behavior (uniquely identified batches climbing to 0.6 vs. near zero for others) and accuracy percentages varying dramatically from 20-100% across runs. While anonymity set sizes show moderate consistency (6-12 range), the substantial differences in privacy outcomes demonstrate that identical system parameters can produce vastly different results depending on random simulation events. This illustrates the challenges in drawing definitive conclusions from limited experimental runs.

3.4 Conclusion

It is important to note that due to computational limitations and the exponential complexity of the batch matching algorithm, our evaluation is based on a limited set of experimental runs. The patterns observed in this study should be interpreted with caution, as they represent only a small sample of the possible parameter space.

The computational complexity of enumerating all valid permutations scales poorly with both the number of clients and batch size, limiting our ability to conduct extensive statistical analysis. Additionally, the single-run approach for most configurations prevents us from establishing statistical significance or confidence intervals for our observations.

Future work should focus on developing more efficient algorithms that can handle larger parameter spaces and enable comprehensive statistical evaluation of anonymity guarantees under various operational conditions.