Batch Mapping in Mixnets

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 only be reconstructed when all sub-messages in the batch are successfully received at the destination. This approach provides fault tolerance 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 fixed-size 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.

2 Proposed Solution: The Mixnet Batch Matching Algorithm

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 while maintaining temporal and batch coherence constraints.

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

This initialization phase sets up the fundamental data structures and identifies eligible messages for batch mapping. The algorithm first initializes empty containers for valid permutations, batch probabilities, and anonymity metrics, then determines which incoming batches are candidates for mapping to the current outgoing batch based on size constraints ($lenIn \geq lenOut$), and finally collects all incoming messages that satisfy the temporal constraint ($t_{M_{ij}} < t_{O_{pq}}$) for subsequent processing.

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 batch, C_p	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 messages 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.

Phase 1: Initialization and Candidate Selection

$Valids \leftarrow []$
$BatchProb \leftarrow \{\}$
$AnonymitySet \leftarrow \{\}$
$AnonymitySetSize \leftarrow \{\}$
Outgoing message O_{pq} enters the outgoing batch, p at time $t_{O_{pq}}$
$OutMsgMappingSet[p] \leftarrow \{\}$
for each i in $IncomingBatches$ **do**
    $lenIn \leftarrow len(IncomingBatches[i])$
    $lenOut \leftarrow len(OutgoingBatches[p])$
    if $lenIn \geq lenOut$ **then**
        $OutBatchMappingCount[p][i] \leftarrow 0$
    end if
end for
for each i in $OutBatchMappingCount[p]$ **do**
    for each $M_{ij}, t_{M_{ij}}$ in $IncomingBatches[i]$ **do**
        if $t_{M_{ij}} < t_{O_{pq}}$ **then**
            $OutMsgMappingSet[O_{pq}].add(M_{ij})$
        end if
    end for
end for

2.2 Initial Permutation Construction

This phase handles the base case when no valid permutations have been established yet. For each eligible incoming message identified in Phase 1, the algorithm creates a new permutation where that message is mapped to the current outgoing batch position, establishing the initial set of valid mappings that will be expanded in subsequent phases.

Phase 2: Initial Permutation Construction

if $Valids = []$ **then**
    for each $M_{ij} \in OutMsgMappingSet[O_{pq}]$ **do**
        $Valids \leftarrow Valids.append(\{p : [M_{ij}]\})$
    end for
end if

2.3 Permutation Extension and Validation

This is the core expansion phase where existing permutations are extended with new message mappings while enforcing batch coherence constraints. The algorithm processes each eligible incoming message and attempts to incorporate it into existing valid permutations through two cases: extending existing sub-permutations (ensuring all messages come from the same incoming batch) or creating new sub-permutations (ensuring the incoming batch hasn't been used elsewhere in the permutation).

Phase 3: Permutation Extension and Validation

```
if Valids ≠ [] then
    tempValids ← []
    for each M_ij ∈ OutMsgMappingSet[O_pq] do
        i ← batchid(M_ij)
        j ← msgid(M_ij)
        for each x in Valids do
            newX ← x
            count ← 0
            msgList ← newX.get(p)
            if msgList exists then
                                                    ▷ Case 1: Extend existing sub-permutation
                for v ← 0 to len(msgList) do
                    b ← batchid(msgList[v])
                    m ← msgid(msgList[v])
                    if b = i and m ≠ j then
                        count ← count + 1
                    else
                        break
                    end if
                end for
                if count = len(msgList) then
                    newMsgList ← appendMsg(msgList, M_ij)
                    newX[p] ← newMsgList
                    tempValids ← tempValids.append(newX)
                    OutBatchMappingCount[p][i] ← OutBatchMappingCount[p][i] + 1
                    count ← 0
                else
                    count ← 0
                end if
            else
                                                    ▷ Case 2: Create new sub-permutation
                for each batchMsgs in x do
                    if batchid(batchMsgs[0]) ≠ i then
                        count ← count + 1
                    end if
                end for
                if count = len(x) then
                    newX[p] ← [M_ij]
                    tempValids ← tempValids.append(newX)
                    OutBatchMappingCount[p][i] ← OutBatchMappingCount[p][i] + 1
                    count ← 0
                else
                    count ← 0
                end if
            end if
        end for
    end for
    Valids ← tempValids
    tempValids ← []
end if
```

2.4 Global Batch Mapping Update

After updating permutations for the current outgoing batch, this phase ensures that mapping counts for all other outgoing batches remain consistent with the new set of valid permutations. For each valid permutation and each outgoing batch position (except the current one), the algorithm identifies which incoming batch contributes to that position and increments the corresponding count, maintaining the integrity of probability calculations across all outgoing batches.

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

The final phase computes normalized probability distributions and anonymity metrics for all outgoing batches by calculating the probability that each outgoing batch originated from each candidate incoming batch. The algorithm constructs anonymity sets containing all incoming batches with non-zero mapping probabilities, computes anonymity set sizes for privacy quantification, and resets data structures to prepare for processing the next outgoing message.

Phase 5: Probability Computation and Finalization

for each $outBatch$ in $OutBatchMappingCount$ **do**
  if $outBatch$ not in $BatchProb$ **then**
    $BatchProb[outBatch] \leftarrow \{\}$
  end if
  $nonZero \leftarrow \{\}$
  for each $(inBatch, count)$ in $OutBatchMappingCount[outBatch]$ **do**
    $prob \leftarrow \frac{count}{len(Valids)}$
    if $prob > 0$ **then**
      $nonZero[inBatch] \leftarrow prob$
    end if
  end for
  if $nonZero$ **then**
    $BatchProb[outBatch] \leftarrow nonZero$
    $AnonymitySet[outBatch] \leftarrow \{nonZero.keys()\}$
    $AnonymitySetSize[outBatch] \leftarrow len(AnonymitySet[outBatch])$
  else
    if $outBatch$ in $BatchProb$ **then**
      $delBatchProb[outBatch]$
    end if
  end if
end for
$OutBatchMappingCount \leftarrow \{\}$
$BatchProb \leftarrow \{\}$
$AnonymitySet \leftarrow \{\}$
$AnonymitySetSize \leftarrow \{\}$

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 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.

3.2 Observations

3.2.1 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, 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.

3.2.2 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. 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.

3.2.3 Reproducibility Analysis

Figure 5 shows multiple runs of the same parameter configuration (20 clients, batch size 4) to assess the reproducibility and variance in our results. 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 reproducibility 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, highlighting the stochastic nature of mixnet behavior and the challenges in drawing definitive conclusions from limited experimental runs.

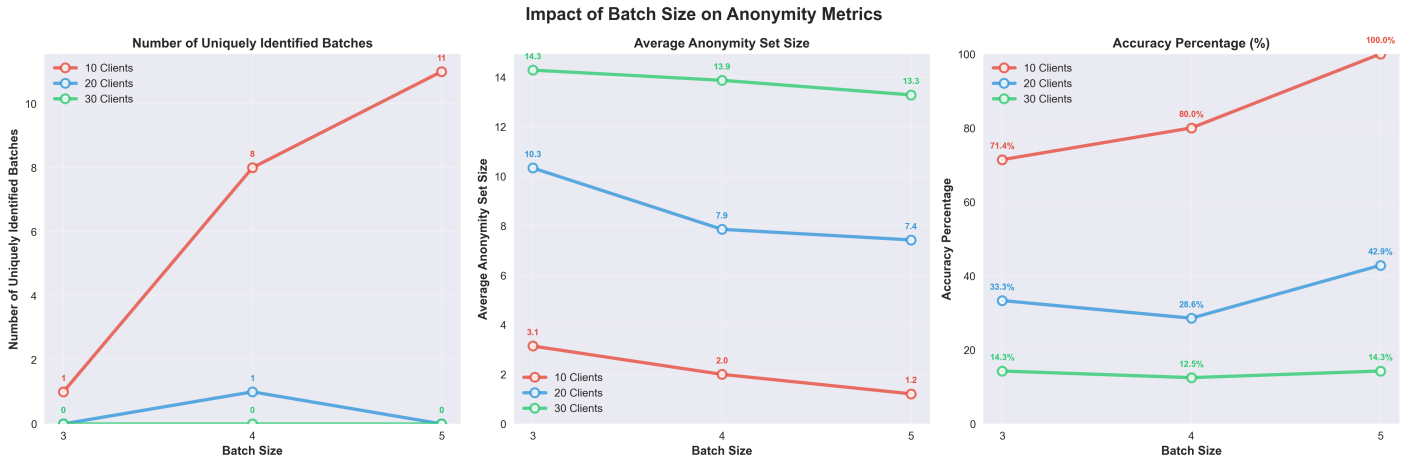

Figure 1: Impact of batch size on various anonymity metrics across different numbers of clients (10, 20, and 30 clients). The plots show how batch size affects the number of uniquely identified batches, average anonymity set size, and accuracy percentage.

3.3 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.

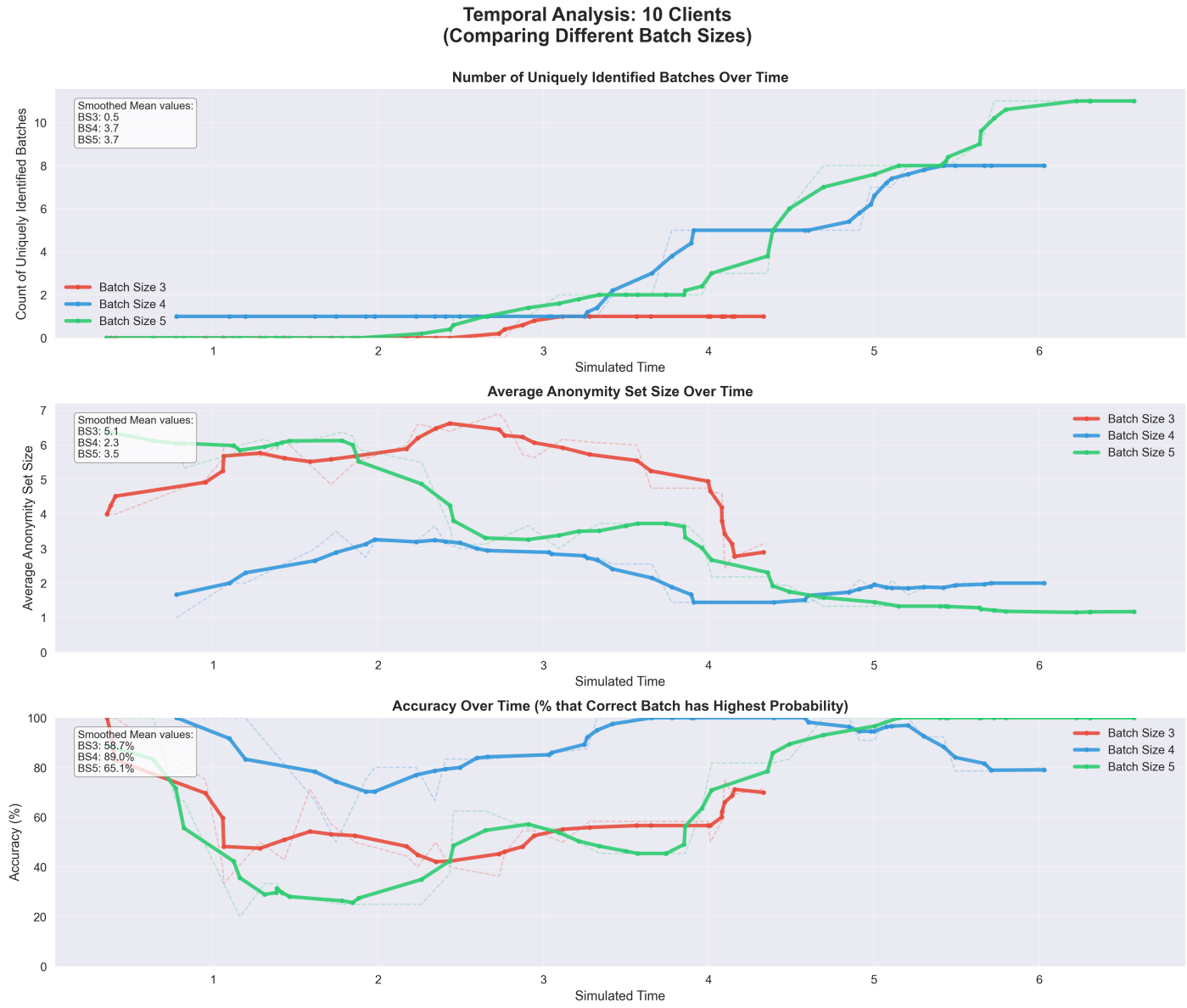

Figure 2: Temporal analysis of anonymity metrics for 10 clients across different batch sizes (3, 4, and 5).

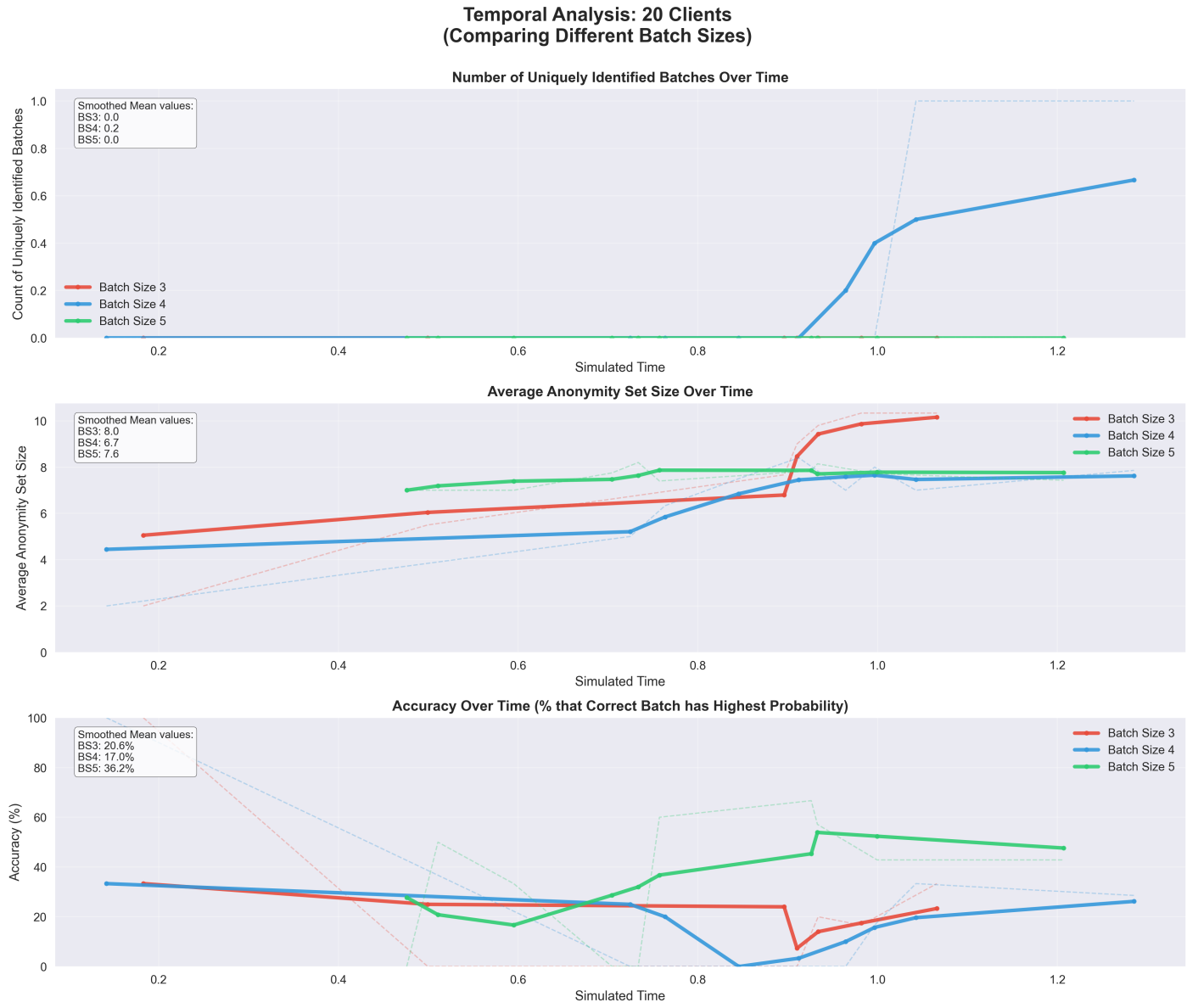

Figure 3: Temporal analysis of anonymity metrics for 20 clients across different batch sizes (3, 4, and 5).

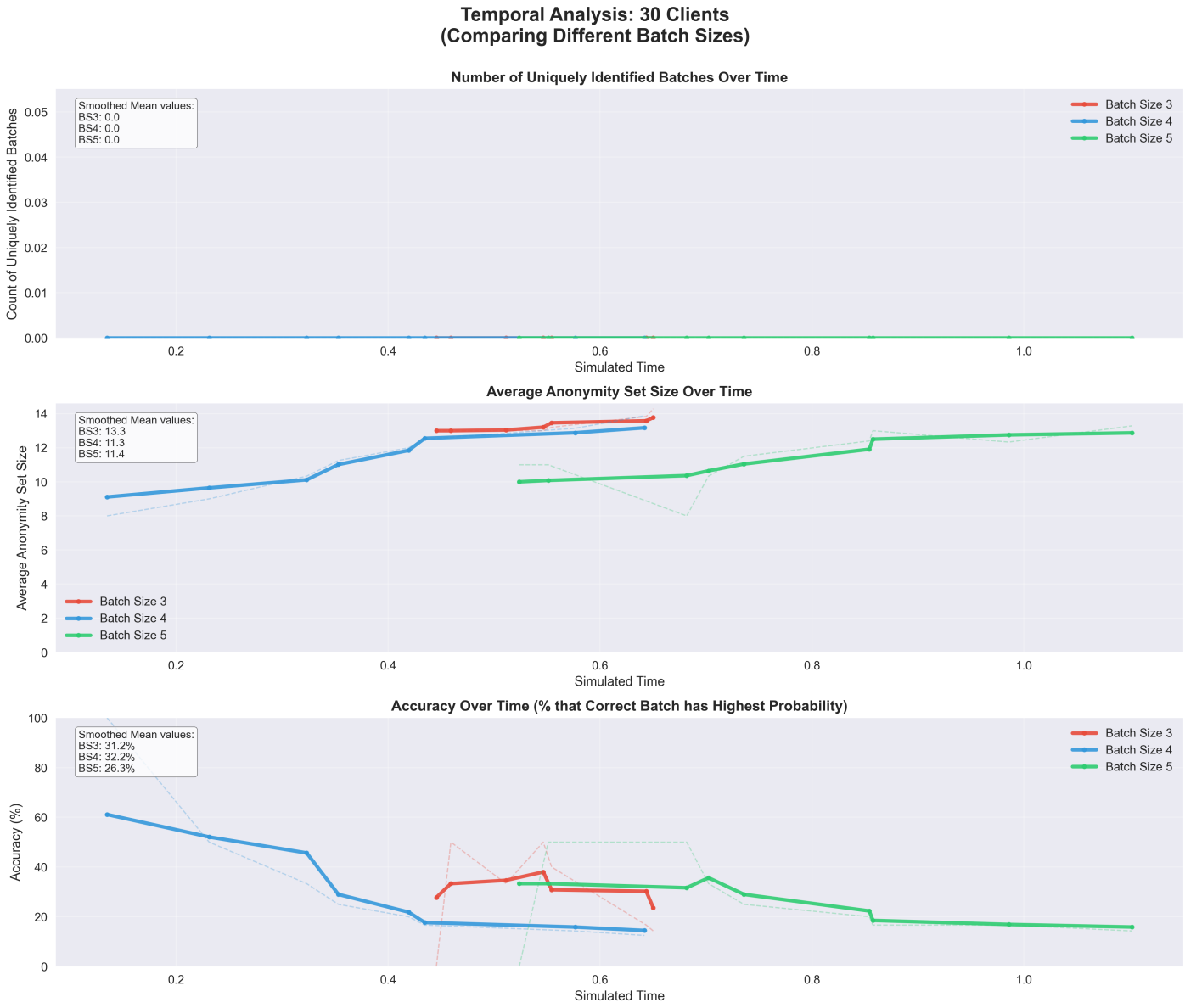

Figure 4: Temporal analysis of anonymity metrics for 30 clients across different batch sizes (3, 4, and 5).

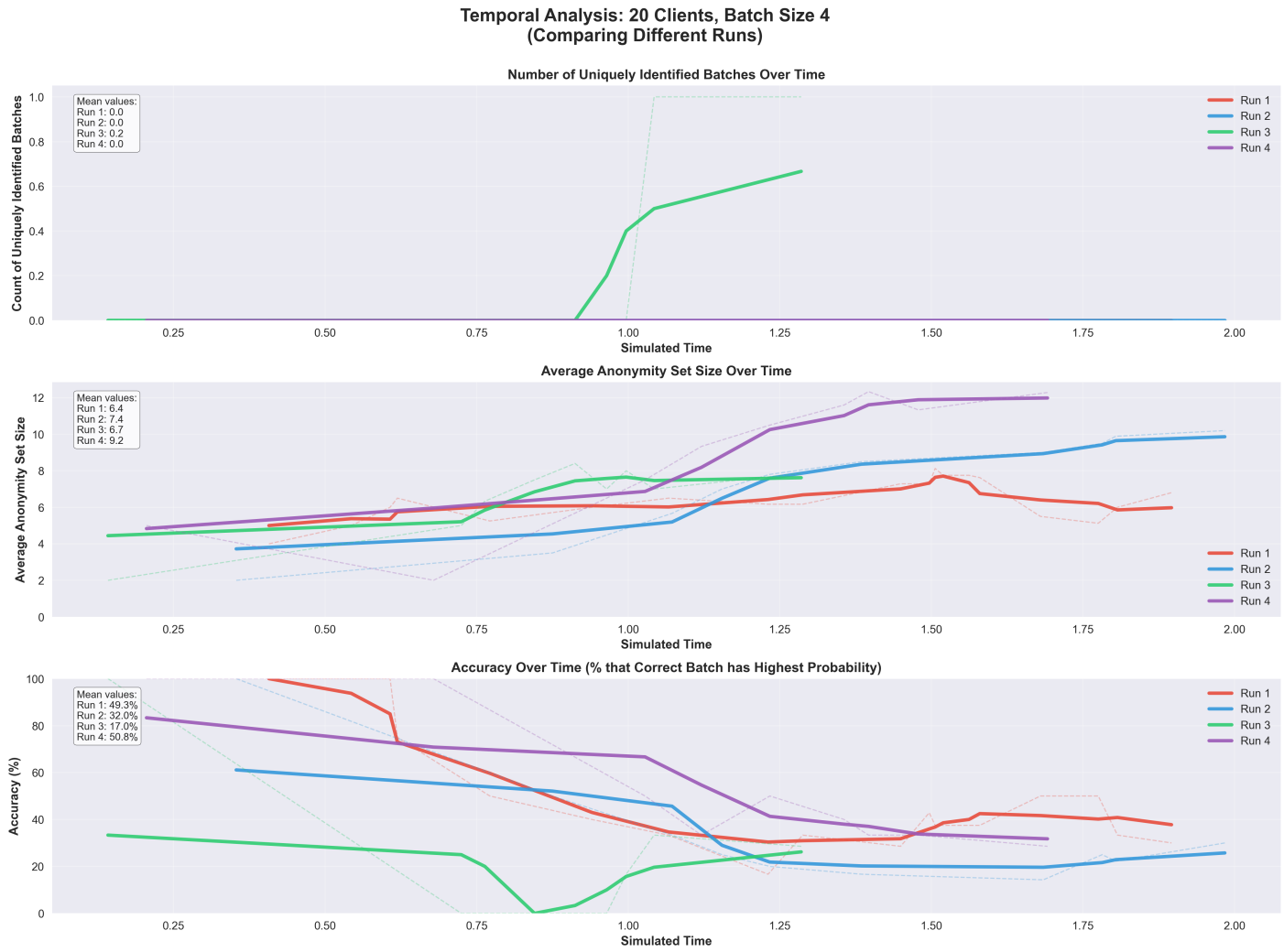

Figure 5: Reproducibility analysis showing temporal evolution of anonymity metrics across four independent simulation runs with identical parameters (20 clients, batch size 4).