Probabilistic Batch Mapping in Mixnets

Aiswarya Walter, Student ID: 427199

1 Problem Definition

Mix networks (mixnets) are cryptographic systems that provide anonymous communication 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 robust 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 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 fundamental privacy challenge arises when a global adversary observes the mixnet system. Such an adversary can monitor all incoming and outgoing traffic at mixing nodes, recording message arrival and departure timestamps, batch compositions, and routing patterns. Given this comprehensive view of network activity, the adversary attempts to perform traffic analysis to determine which incoming message batches correspond to which outgoing message batches.

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, we aim 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 following describes each major stage of the Mixim Batch Matching algorithm, corresponding to the pseudocode steps.

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	Dict of set of messages
$Valids$	List of valid message permutations for all outgoing batches	List of dict of lists
$BatchProb$	Probability mapping for all output batches	Dict of dict
$batchid(msg)$	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.

2.1 Phase 1: Initialization and Candidate Selection

Algorithm 1 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: $OutMsgMappingSet[p] \leftarrow \{\}$
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**

Phase 1 Explanation: 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 ($Valids$), batch probabilities ($BatchProb$), and anonymity metrics (Lines 1-4). When a new outgoing message O_{pq} arrives at time $t_{O_{pq}}$, the algorithm determines which incoming batches are candidates for mapping to the current outgoing batch C_p based on size constraints (Lines 7-12). Only incoming batches with sufficient messages ($lenIn \geq lenOut$) are considered viable candidates. Finally, the algorithm collects all incoming messages that satisfy the temporal constraint ($t_{M_{ij}} < t_{O_{pq}}$) and belong to candidate batches, storing them in $OutMsgMappingSet$ for subsequent processing (Lines 13-19).

i. Initialize Candidate Sets. The algorithm begins by initializing empty structures for valid permutations ($Valids = []$) and batch probabilities ($BatchProb = \{\}$) (Lines 1–2). When an outgoing message O_{pq} enters the outgoing batch C_p, $BatchMappingC_p$ is initialised to an empty dictionary so that the count of valid permutations can be recalculated (Lines 3-4). It then determines which incoming batches are eligible to have contributed to the current outgoing batch C_p (Lines 5–9). Specifically, for each incoming batch, if the number of messages is greater than or equal to the number of messages in the outgoing batch C_p, an entry is added to $BatchMappingC_p$ with an initial count of 0. Subsequently, all messages M_{ij} that arrived before O_{pq} and belong to a batch already present in $BatchMappingC_p$ are collected in $MsgMappingO_{pq}$ (Line 10).

2.2 Phase 2: Initial Permutation Construction

Algorithm 2 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**

Phase 2 Explanation: This phase handles the base case when no valid permutations have been established yet. For each eligible incoming message M_{ij} identified in Phase 1, the algorithm creates a new permutation where that message is mapped to the current outgoing batch position p. Each permutation is represented as a dictionary mapping outgoing batch indices to lists of incoming messages. This establishes the initial set of valid mappings that will be expanded in subsequent phases.

ii. Construct Initial Valid Permutations. If no valid permutations exist yet (i.e., $Valids = []$) (Line 11), each eligible message M_{ij} from $MsgMappingO_{pq}$ is wrapped in its own single-message path and added to $Valids$ (Lines 12–14). This forms the initial set of valid permutations.

2.3 Phase 3: Permutation Extension and Validation

Algorithm 3 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 inValids 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
                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
                end if
            end if
        end for
    end for
    Valids ← tempValids
end if
```

Phase 3 Explanation: This is the core expansion phase where existing permutations are extended with new message mappings. The algorithm processes each eligible incoming message M_{ij} and attempts to incorporate it into each existing valid permutation. Two distinct cases are handled: (1) **Extending existing sub-permutations**: If the current permutation already has messages mapped to outgoing batch p, the algorithm verifies that all existing messages come from the same incoming batch i and that M_{ij} is distinct from them. If valid, M_{ij} is appended to the existing sub-permutation. (2) **Creating new sub-permutations**: If no messages are yet mapped to batch p, the algorithm ensures that incoming batch i hasn't been used elsewhere in the permutation before creating a new sub-permutation containing only M_{ij}. This phase enforces the critical constraint that messages from the same incoming batch must map to the same outgoing batch, maintaining batch coherence.

iii. Expand Valid Permutations. If $Valids$ already contains existing permutations (Lines 15–55), the algorithm attempts to extend each path with the current message M_{ij}. This is handled in two cases:

- **Existing sub-permutation for batch p:** If the current permutation already has a sub-permutation at index p (Lines 20–37), the algorithm checks whether all messages in that sub-permutation originate from batch B_i and are distinct from M_{ij}. If so, M_{ij} is appended to that sub-permutation, and the updated permutation is added to $tempValids$. The corresponding entry in $BatchMappingC_p$ is incremented.
- **No existing sub-permutation for batch p:** Otherwise (Lines 38–52), the algorithm verifies that batch B_i is not already present in earlier sub-permutations. If valid, a new sub-permutation containing M_{ij} is appended to the path, and the updated path is added to $tempValids$. The batch count in $BatchMappingC_p$ is incremented.

At the end of this step, $tempValids$ replaces $Valids$ (Line 56).

2.4 Phase 4: Global Batch Mapping Update

Algorithm 4 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

Phase 4 Explanation: After updating permutations for the current outgoing batch C_p, 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 in $OutBatchMappingCount$. This global update maintains the integrity of probability calculations across all outgoing batches, ensuring that changes to one batch's permutations are properly reflected in the mapping statistics of all other batches.

iv. Update Batch Mappings for All Other Outgoing Batches. For each valid path in $Valids$, the algorithm iterates through all output batches r except the one corresponding to the current batch $(p-1)$ (Lines 57–64). For each such position, the incoming batch is identified using `batchid()`, and the corresponding count in $BatchMappingC_{r+1}$ is incremented. This ensures that, in addition to updating counts for the current output batch C_p, the algorithm also maintains counts for all other outgoing batches in the permutation.

2.5 Phase 5: Probability Computation and Finalization

Algorithm 5 Phase 5: Probability Computation and Finalization

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

Phase 5 Explanation: The final phase computes normalized probability distributions and anonymity metrics for all outgoing batches. For each outgoing batch, the algorithm calculates the probability that it originated from each candidate incoming batch by dividing the count of valid permutations by the total number of valid permutations ($|Valids|$). Only non-zero probabilities are retained in the final *BatchProb* mapping. Simultaneously, the algorithm constructs anonymity sets containing all incoming batches with non-zero mapping probabilities and computes the anonymity set size, which quantifies the privacy protection level for each outgoing batch. Batches with no valid mappings are removed from the probability distribution. Finally, the algorithm resets temporary data structures to prepare for processing the next outgoing message.

v. Compute Batch Probabilities for All Outgoing Batches. Finally, the algorithm computes normalized probabilities for every outgoing batch (Lines 65–69). For each index r (covering all batches in each valid permutation) and each incoming $(batch, count)$ in $BatchMappingC_{r+1}$, the probability is calculated.

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.

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.

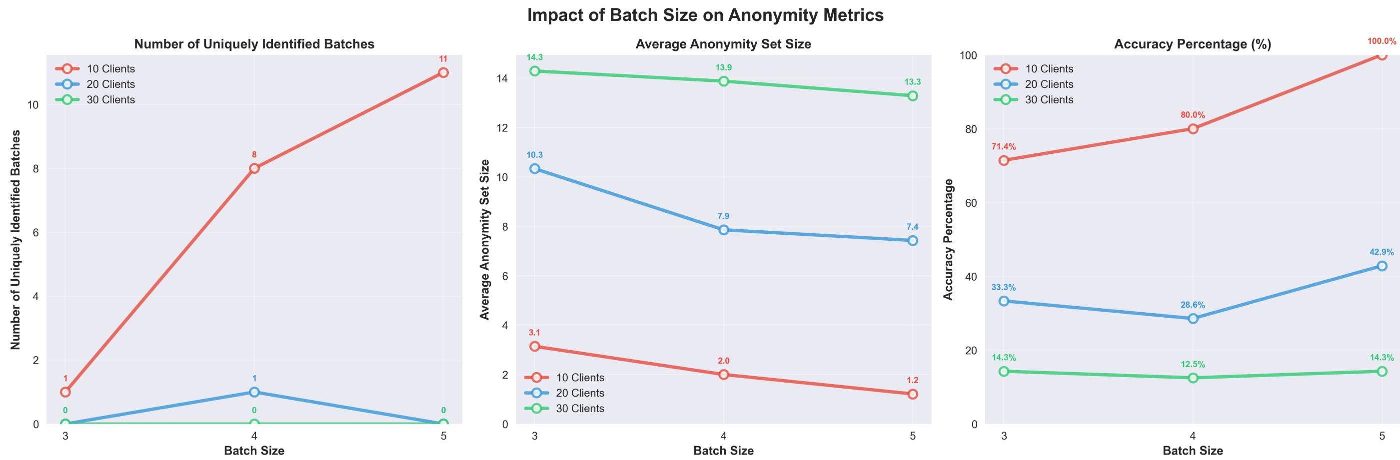

Figure 1: Impact of batch size on various anonymity metrics across different numbers of clients (10, 20, and 30 clients). The plots show how batch size affects the number of uniquely identified batches, average anonymity set size, and accuracy percentage.

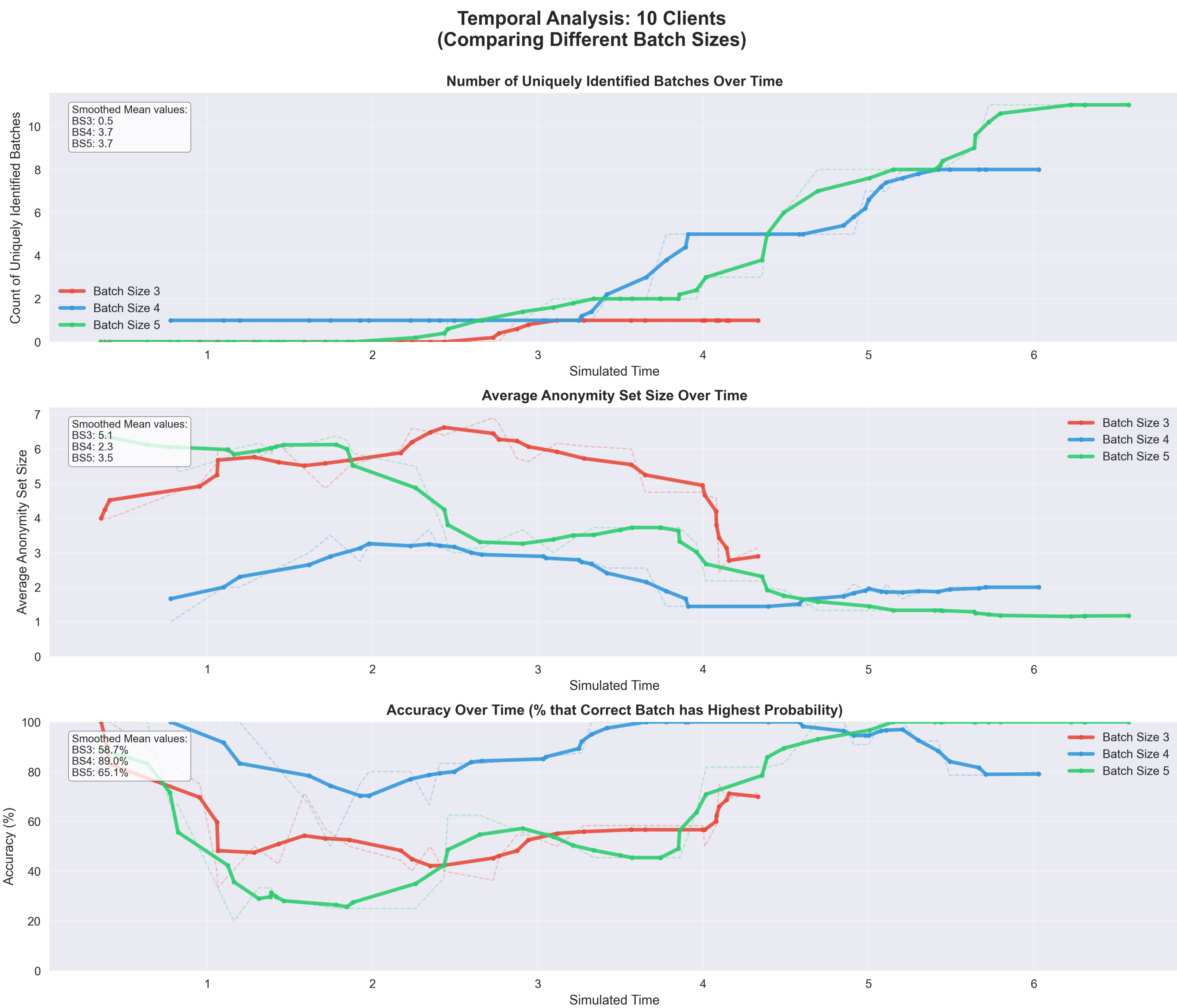

Figure 2: Impact of batch sizes when number of clients is 10

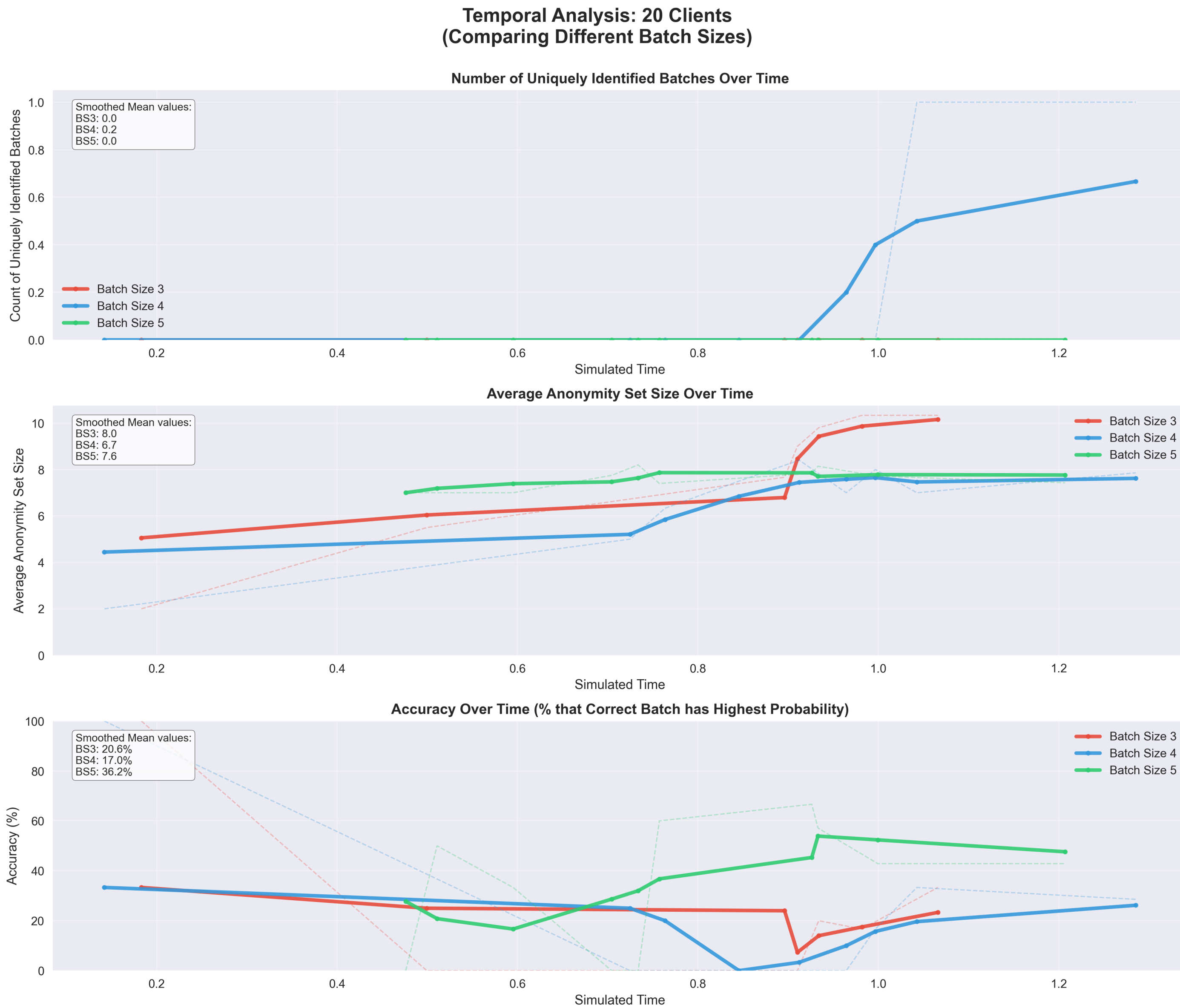

Figure 3: Impact of batch sizes when number of clients is 20

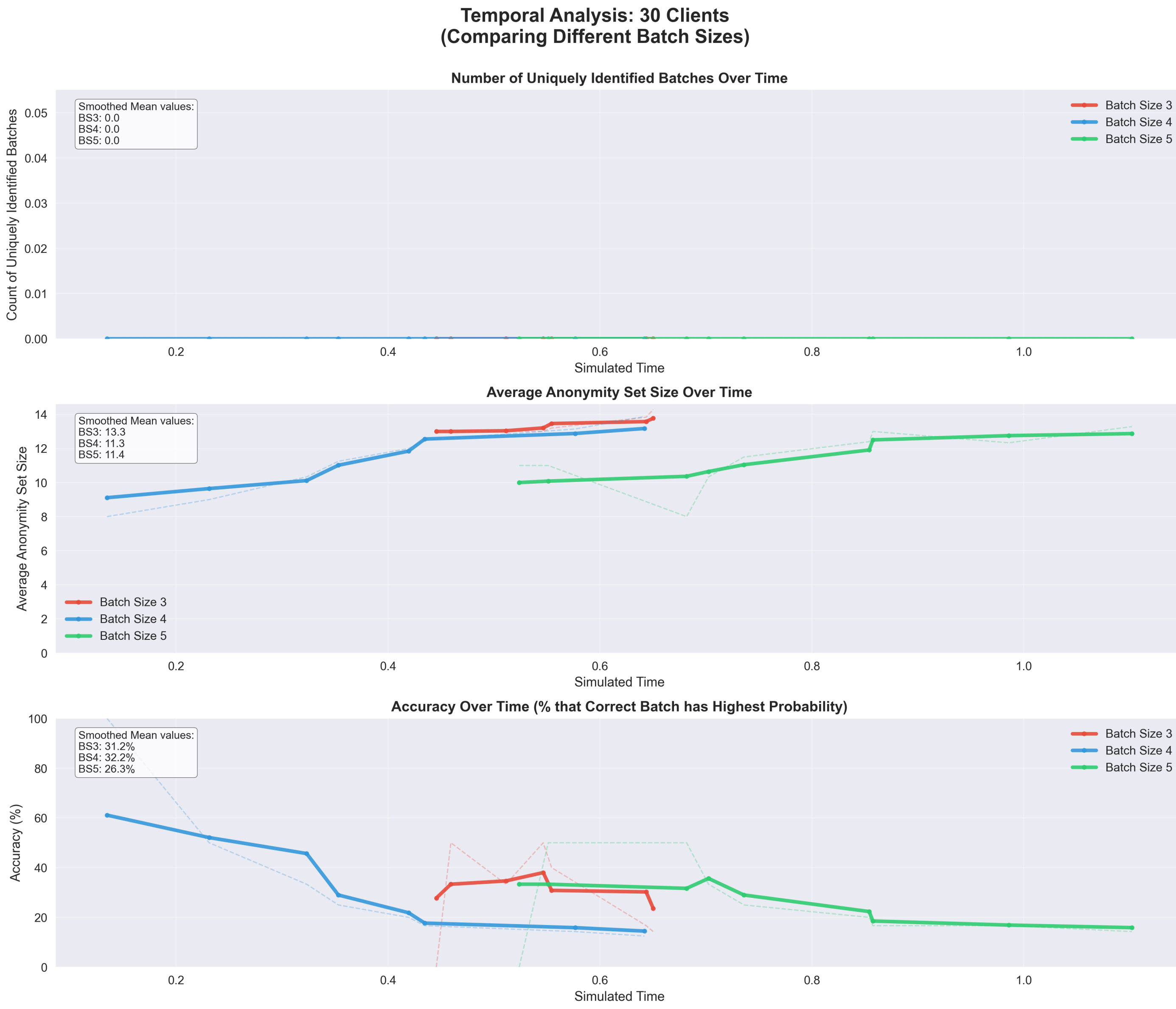

Figure 4: Impact of batch sizes when number of clients is 30

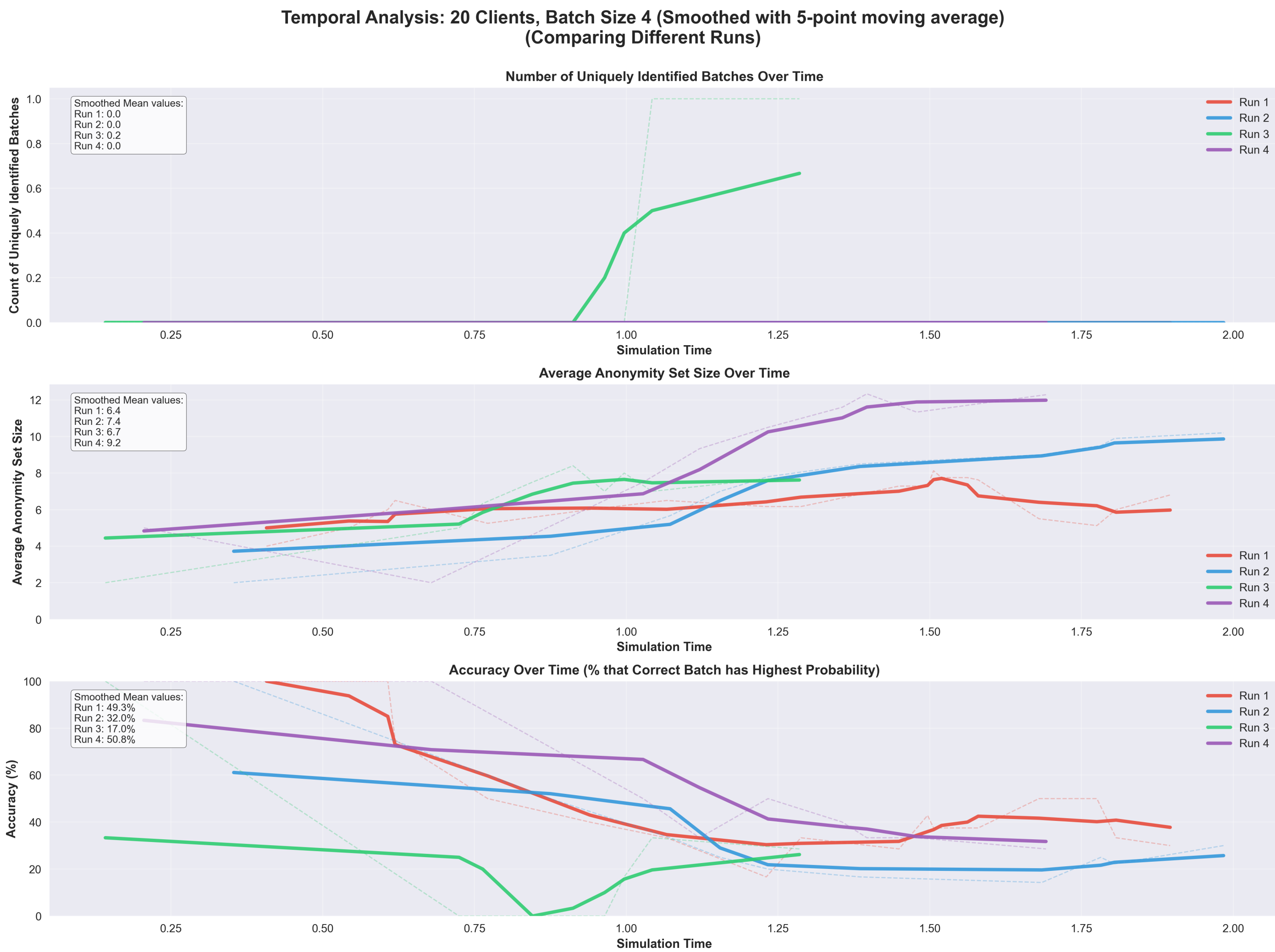

Figure 5: Comparison of 20 client 4 batch runs