On Frontrunning Risks in Batch-Order Fair Systems for Blockchains

(Extended Version)*

Eunchan Park†
paul.park@kaist.ac.kr
KAIST

Taeung Yoon†
yoontaeung@kaist.ac.kr
KAIST

Hocheol Nam
hcnam@kaist.ac.kr
KAIST

Deepak Maram
deepak@mystenlabs.com
Mysten Labs

Min Suk Kang‡
minsukk@kaist.ac.kr
KAIST

ABSTRACT

In timing-sensitive blockchain applications, such as decentralized finance (DeFi), achieving first-come-first-served (FCFS) transaction ordering among decentralized nodes is critical to prevent frontrunning attacks. Themis [30], a state-of-the-art decentralized FCFS ordering system, has become a key reference point for high-throughput fair ordering systems for real-world blockchain applications, such as rollup chains and decentralized sequencing, and has influenced the design of several subsequent proposals. In this paper, we critically analyze its core system property of practical batch-order fairness and evaluate the frontrunning resistance claim of Themis. We present the Ambush attack, a new frontrunning technique that achieves nearly 100% success against the practical batch-order fair system with only a single malicious node and negligible attack costs. This attack causes a subtle temporary information asymmetry among nodes, which is allowed due to the heavily optimized communication model of the system. A fundamental trade-off we identify is a challenge in balancing security and performance in these systems; namely, enforcing timely dissemination of transaction information among nodes (to mitigate frontrunning) can easily lead to non-negligible network overheads (thus, degrading overall throughput performance). We show that it is yet possible to balance these two by delaying transaction dissemination to a certain tolerable level for frontrunning mitigation while maintaining high throughput. Our evaluation demonstrates that the proposed delayed gossiping mechanism can be seamlessly integrated into existing systems with only minimal changes.

1 INTRODUCTION

Consensus on the order of transactions is a fundamental requirement for blockchain systems. This is particularly crucial for recent blockchain applications, such as decentralized finance (DeFi) and decentralized sequencing. These applications emphasize the importance of ordering at much finer granularity, such as the relative ordering of two adjacent transactions within a block. Minor alterations in transaction order by a malicious actor within a blockchain system have been shown to significantly benefit the perpetrators. One notable form of attack is *frontrunning* [18, 20, 48], where a perpetrator receives a transaction (which becomes the target of the attack) from a user, quickly creates a frontrunning transaction based on the target transaction, and places it *before* the target transaction. Studies have documented that frontrunning attacks have been used to gain significant economic benefits in blockchains [42, 43, 48].

A key property for mitigating frontrunning attacks is the first-come-first-served (FCFS) ordering policy for user transactions. This ensures that a frontrunning transaction generated after a target transaction cannot be possibly processed earlier. For blockchain systems, FCFS ordering is desired in a decentralized manner, where multiple nodes independently order received transactions and then agree on the final order. This prevents any single node from manipulating the transaction sequence for its benefit. Decentralized FCFS ordering has been widely studied in recent years [1, 7, 14, 17, 30–32, 36, 37, 40, 54, 56]. Among them, Kelkar et al. [31] introduced the first formal notion of *fair ordering* in a decentralized setting; i.e., guaranteeing that transaction A is sequenced before transaction B if the majority of nodes observe A first [31]. Specifically, their proposed *batch-order fairness* [31] ensures strong FCFS ordering, except when a set of transactions form a Condorcet cycle [16], making it infeasible for the majority to determine their order within the cycle.

While the concept of batch-order fairness is theoretically sound and effective against frontrunning attacks, its initial design [31] is too costly for practical use. A recent advancement, Themis [30], has significantly improved the efficiency of batch-order fairness, making it practical (hereafter referred to as the *practical batch-order fair* system) for real-world applications, such as layer-2 rollups. Themis' leader-based network design achieves quadratic message complexity, making it the first decentralized fair ordering scheme suitable for commercial applications with high throughput requirements. Beyond performance, Themis [30] also provides a stronger liveness guarantee than its predecessor [31] and offers a security guarantee against frontrunning attacks when up to f replicas are Byzantine among a network of $n \geq 4f + 1$ replicas. This practical batch-order fair system has been a key reference point (often as a boilerplate) for many subsequent decentralized fair ordering systems; see how SpeedyFair [40], Rashnu [1], Quick Order Fairness [14], and Auncel [15] have improved upon Themis [30] (refer to Section 2.2 for more details). Moreover, Themis had been considered as a potential solution for the decentralized sequencing in major rollups, such as Arbitrum [34], and oracle networks, such as Chainlink [28] (until recently they decided to adopt hybrid sequencing models for better revenue generation [8, 33]).

In this paper, we critically examine the state-of-the-art practical batch-order fair system, focusing on the subtle trade-offs between

*A conference version of this paper appears in the Proceedings of the 2025 ACM Conference on Computer and Communications Security (CCS).
†Both authors contributed equally to this research.
‡Corresponding author.

its frontrunning resistance and throughput performance. We first present a new attack, which we name the Ambush attack, that shows that even a single malicious replica can reliably achieve near 100% success in frontrunning any target transaction by simply submitting a few additional transactions before and after the target transaction with carefully timed intervals for each replica. The proposed attack demonstrates that the frontrunning resistance claim of the practical batch-order fair system is flawed due to the artifacts in the design of its current underlying network model. In particular, we discover that the Ambush attack is highly effective because of the optimized transaction information dissemination between replicas, which is a key design choice for achieving high throughput in the practical batch-order fair system. Without proper dissemination among replicas, an adversary can easily create information asymmetry among them, enabling highly effective frontrunning attacks.

To that end, we propose a simple, practical countermeasure that nicely balances undesirable trade-offs between security and performance in the practical batch-order fair system. The proposed *delayed gossiping mechanism* probabilistically disseminates user transactions among replicas, effectively reducing information asymmetry and significantly reducing the effectiveness of the Ambush attack while only incurring marginal message overhead. The key to designing such a countermeasure is to understand how much temporal information asymmetry is tolerable for the system to prevent the Ambush attack successfully; that is, the system may allow a certain level of information asymmetry while mitigating the majority of frontrunning attacks. To break information asymmetry among replicas with some delays, the system groups information dissemination, thus maintaining the network performance. Our proposed delayed gossiping mechanism is shown to achieve a sweet spot per-transaction delay for gossiping that achieves the desired balance. The scheme is not only easy to implement but also simple to operate as well because it is tuned by two security parameters, gossip probability p and group size s, which can be dynamically adjusted based on the system's security requirements, network conditions, and performance constraints.

Beyond the specific frontrunning strategies and corresponding countermeasures, our work offers a valuable lesson for designing decentralized fair ordering systems. Particularly, it underscores the importance of understanding how nuanced network design choices can influence critical security properties in distributed algorithms for blockchain systems.

In this paper, we make the following contributions:

- We present a new frontrunning attack, named the Ambush attack, against the state-of-the-art practical batch-order fair system, demonstrating near 100% success in various practical scenarios (Section 4 and 5).
- We investigate the root cause of the discovered vulnerability, which resides in the subtle network protocol design of the target system. We provide empirical evidence of the resulting trade-off, highlighting the challenge of achieving both security and performance in practice (Section 6).
- We propose a practical countermeasure, the delayed gossiping mechanism, which nicely balances with the trade-offs by mitigating the Ambush attack effectively while keeping the performance degradation minimal (Section 7).

2 BACKGROUND

We provide background on several important notions, starting from frontrunning attacks to fair ordering in blockchain systems.

2.1 Frontrunning Attacks

Frontrunning [18, 20, 42, 43, 48] is a widely studied attack technique often used in maximal extractable value (MEV) attacks [18, 22, 24, 42, 43, 50, 58]. It is a transaction reordering attack that exploits the (partial) knowledge of a victim's transaction to execute a trade ahead of it. In decentralized exchanges (DEXes), where trades are executed against liquidity pools, frontrunners can generate significant profits by exploiting price movements caused by the victim's trade. By executing their own trade before the victim's, frontrunners can either push prices unfavorably for the victim or profit from the price changes triggered by the victim's trade. Several measurement studies have independently identified thousands of frontrunning attacks in the wild over different periods; e.g., McLaughlin et al. found 63,257 frontrunning incidents (about 5.2 billion USD) over 29 months [39], Torres et al. found 199,725 incidents (about 18.4 million USD) over 64 months [48], and Qin et al. found 750,529 incidents (174.3 million USD) over 32 months [43].

In sandwich attacks, frontrunning is combined with backrunning (i.e., executing a trade after the victim's transaction) to further exploit price discrepancies caused by the victim's trade. For example, McLaughlin et al. [39] present one simple sandwich attack, where an attacker initially buys 38,122 DAI with 298,613 HOPR on Uniswap v3 before the victim buying 63,139 DAI with 500,000 HOPR, and subsequently sells all 38,122 DAI for 300,868 HOPR afterwards, earning a profit of $59. In this paper, we focus exclusively on frontrunning attacks, as backrunning is generally easier (or trivial in some cases) compared to frontrunning.

2.2 Fair Ordering

To mitigate frontrunning attacks (or order manipulation more broadly) by malicious operators, numerous studies have been conducted under the umbrella of *fair ordering*. While the precise definition of fair ordering varies depending on the system design and requirements, the shared objective is to approximate a first-come-first-served (FCFS) transaction order. In essence, transactions generated earlier should be ordered before those generated later.

One approach to achieving fair ordering is to conceal transactions from the system until a certain point in time when their order is finalized and confirmed [12, 13, 47]. However, this approach requires additional cryptographic operations and key exchanges, which can slow down transaction processing [25]. A related approach is to conduct sequencing operations in a trusted execution environment (TEE) [11, 41, 55] to prevent manipulation. However, this approach relies on trusted hardware at the blockchain nodes and imposes additional costs for user-initiated remote attestation [25].

Decentralized FCFS ordering. A promising alternative, which we focus on in this work, is the use of a decentralized committee of replicas to order transactions in a first-come-first-served manner. With the threat model of certain number of Byzantine replicas within the committee, this approach ensures that no single or a few replicas can manipulate the order of transactions. Several recent studies [1, 7, 14, 17, 30–32, 36, 37, 40, 45, 54, 56] have explored ordering systems for decentralized committees. As there exist no reliable way to measure the exact time of transaction generation, these systems rely on the receive time of transactions at each replica to order them, which is called receive-order fairness. How each system records and reports the receive times of transactions, and reaches consensus on the final order, may vary across different systems. For a comprehensive overview, refer to Baum et al. [9] and Heimbach et al. [25].

Batch-order fair systems. The first formal approach to fair ordering was proposed by Kelkar et al. [31], who reviewed various notions of fairness, including send-order fairness and receive-order fairness, and discussed their limitations. Notably, they identified that receive-order fairness is infeasible to achieve in real-world setting due to the Condorcet paradox [16]. This paradox occurs when a cycle exists in pairwise ordering. For instance, if three replicas report receiving transactions t_A, t_B, t_C in the order $t_A \to t_B \to t_C$, $t_B \to t_C \to t_A$, and $t_C \to t_A \to t_B$, respectively, a Condorcet cycle ($t_A \to t_B \to t_C \to t_A$) is formed, making receive-order fairness unattainable.

Acknowledging the Condorcet paradox, Kelkar et al. [31] proposed the concept of γ-*batch-order fairness*, which ensures that if a γ-fraction of replicas agree on the order of two transactions, those transactions are ordered accordingly. The protocol ensures that any two transactions are ordered based on the γ-majority agreement of replicas, except for transactions within a batch, which may not be ordered fairly. This original protocol also guarantees weak-liveness and safety within a permissioned committee containing at most f faulty replicas in a network of $n \geq 4f + 1$. However, its reliance on broadcast communication among all replicas, using the first-come-first-served broadcast primitive (FCFS-BC) [27], makes it impractical for real-world high-throughput applications.

The original batch-order fairness protocol [31] was enhanced in subsequent work, Themis [30], to make it more practical for high-throughput blockchain applications. Themis employs a leader-based consensus protocol, such as HotStuff [52], to achieve the same γ-batch-order fairness guarantees but with a more efficient communication model. Specifically, replicas independently observe user transactions (*without* broadcasting them) and report their observations to a leader, which aggregates the reports and proposes a final transaction order. This design reduces the communication complexity to $O(n^2)$, compared to the original protocol's $O(n^3)$, where n is the number of replicas. With this improved communication complexity, Themis [30] can handle over a thousand transactions per second, making it the first *practical batch-order fairness* protocol.

Themis [30] (or the broader concept of batch-order fairness [31]) has inspired numerous recent academic works as a key reference point and benchmark for fair ordering in blockchain systems. For instance, SpeedyFair [40] refines Themis by introducing improved pipelining for sequencing and consensus; Rashnu [1] enhances

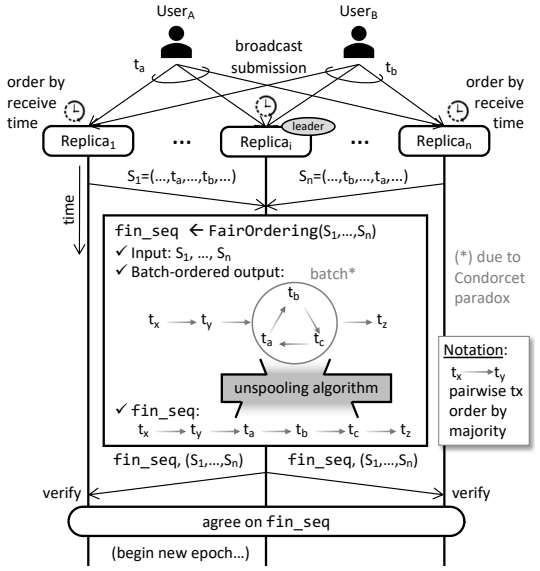


Figure 1: State-of-the-art practical batch-order fair system model [30]

performance by grouping and ordering transactions that access the same data objects; and Quick Order Fairness [14] relaxes the assumption and tolerates up to f faulty replicas with $n \geq 3f + 1$. Another recent fair ordering system, Auncel [15], modifies batch-order fairness by ordering transactions based on per-transaction weights, thereby improving performance without sacrificing the fairness. Evidently, examining the trade-offs between security and performance in Themis remains a critical area of investigation.

While Themis is layer-agnostic (i.e., applicable to both layer-1 and layer-2 systems with minimal design changes, as noted in its original publications [30]), beyond academia, Themis has notably been considered a primary decentralized sequencing solution in major layer-2 rollup projects, such as Arbitrum [21], and orcale networks, such as Chainlink [28]. Early designs of these projects rely on centralized system models, as opposed to the heavily decentralized network structure in layer-1 systems [35, 57]. Their paramount interest has been decentralizing their currently centralized sequencers to resist frontrunning attacks by system operators. Recently, these projects have adopted hybrid sequencing models (e.g., [8, 33]) for their upcoming decentralized sequencing; e.g., Timeboost [33] in Arbitrum allows certain user transactions to be processed earlier through an "express lane" in exchange for payment to the operators. While operating such a hybrid FCFS model is sensible in a highly competitive market, Themis (and its successor projects) remains a reference system for pure FCFS ordering in many real-world blockchain systems (e.g. [3]).

3 SYSTEM AND THREAT MODELS

Let us first summarize the system model that specifies the practical batch-order fair system we consider in this work and define the threat model of the presented frontrunning attacks.

3.1 System Model

The *practical batch-order fair system* we consider in this work takes user transactions by direct submission to a set of n replicas and the leader replica outputs a final sequence of transactions for the time epoch. We reference this common system model from Themis [30], which has inspired many other fair-ordering systems, e.g., Speedy-Fair [40], Rashnu [1], Quick Order Fairness [14], and Auncel [15]. Figure 1 illustrates this system model.

Broadcast-submission model. A key network design feature found in all practical batch-order fair systems [1, 15, 30, 40] is the *broadcast-submission* model. In this model, users are required to submit their transactions *directly to all* replicas in the system, as opposed to the more common gossip-based mempool model employed by many layer-1 blockchain systems. In the gossip-based mempool approach, user transactions are initially submitted to at least one replica and then propagated to other replicas through gossiping.

The broadcast-submission model is specifically designed to ensure fair ordering by allowing each replica to *independently* observe the transaction order without being influenced by other potentially dishonest replicas. In contrast, if replicas rely on gossiping to share transactions (not directly from users), the observed transaction order at one replica could be manipulated by others; e.g., by arbitrarily delaying the propagation of certain user transactions.

Furthermore, gossip-based propagation introduces significant delays, as transactions may need to travel through multiple hops among replicas. This results in highly varying transaction receive times across replicas in an uncontrollable manner, potentially undermining fair ordering. The issue becomes particularly severe in high-throughput systems, where the time gap between consecutive transactions can be much smaller than the propagation delays introduced by gossiping.

The operation of the broadcast-submission model is straightforward: each user submits a new transaction to all replicas in the system, and each replica observes the transaction receive time independently; as seen in Figure 1. With this receive time, an observation of the transaction order that is free from other replicas' influence, each replica can build a local sequence of transactions (e.g., S_i's in the figure), enabling fair and consistent ordering across the system.

Local sequence. Each replica can generate its own sequence of transactions in the order it receives them directly from the users and can report to the leader. When there exist total M transactions submitted during an epoch, $T = \{t_1, t_2, \ldots, t_M\}$, each replica reports its received sequence of transactions, a permutation $S_i \in \text{Perm}(T)$. Figure 1 shows that each non-leader replica reports its local sequence S_i *directly* to the leader.

Fair ordering for batch-order fairness. The leader replica makes the final output `fin_seq` of transactions in the epoch by evaluating `FairOrdering`$(S_1, \cdots, S_n)$, which is a deterministic function that takes the local sequences. The fair-ordering function `FairOrdering()` is the core of the state-of-the-art fair-ordering systems [30, 31, 40] and we follow their common specification and implementation in this work. As shown in Figure 1, `FairOrdering()` guarantees the fair ordering between transactions and some *batches* of transactions; see the batch-ordered output in the figure. The batch-order fairness guarantee ensures that two transactions connected through an arrow (e.g., $t_x \rightarrow t_y$) in the batch-ordered output reflect that the pairwise ordering observed by the majority of replicas. When fair ordering is impossible among a set of transactions (e.g., t_a, t_b, and t_c form a Condorcet cycle), the system batches them and completes the fair ordering with other transactions.

Sequence finalization via batch unspooling. While the batch-ordered output faithfully reflects the pairwise ordering by the majority, it may *not* yet be finalized when it contains one or more cycles in it. For finalization, Themis [30] and other systems [31, 40] *unspool* the Condorcet cycle and conclude the order of transactions within each batch, thereby producing the final sequence `fin_seq`. The unspooling algorithm may vary among different fair-ordering systems, but three important criteria are that (1) the algorithm is deterministic and verifiable by other replicas, (2) the unspooled sequence reflects the pairwise orderings of two adjacent transactions by the majority of the replicas, and (3) the algorithm is efficient to run with a large number (e.g., thousands or more) of transactions in a batch. Satisfying the above criteria, Themis [30] uses a deterministic algorithm that first searches for a Hamiltonian cycle in the given batch and *only then* outputs one Hamiltonian path by rotating the cycle, producing a single ordering of batched transactions. Appendix C provides additional details about this deterministic algorithm.

3.2 Threat Model

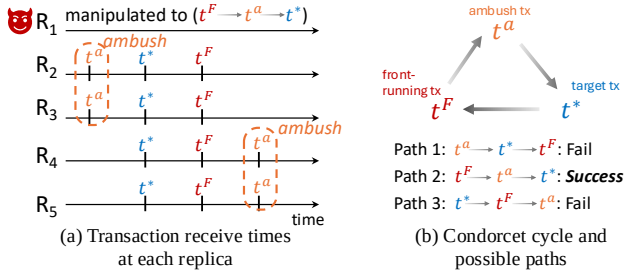
We consider a frontrunning attack scenario against a practical batch-order fair system that tolerates up to f Byzantine replicas. We assume that at least one of these Byzantine replicas attempts to frontrun a specific target transaction. This reflects a practical scenario in blockchain systems, where a system operator or validator (or multiple) attempts to manipulate transaction ordering to their advantage; e.g., a malicious sequencer node (among a committee of decentralized sequencers) of a layer-2 rollup chain may seek to frontrun another transaction.

For simplicity, we assume throughout this paper that only a single adversary replica (i.e., one of the f Byzantine replicas) attempts to frontrun, unless otherwise specified. Frontrunning attacks involving multiple (up to f) colluding replicas are also possible, as discussed further in Section 7.[1]

The goal of the attack is to frontrun a target transaction; that is, an adversary creates a new transaction, called the frontrunning transaction, after observing the target transaction, and ensuring that the frontrunning transaction is placed before the target in the final sequence.

The adversary has full control over its own replica: it can monitor its own transaction queue and observe the transactions submitted by other users. The adversary can also create and control one or more user accounts in the system to submit some adversary-generated user transactions to the non-adversary replicas. The adversary-generated transactions include the frontrunning transaction and some auxiliary transactions that are used to enable frontrunning and enhance it. We assume that the adversary is able to pay a small operational cost for creating and submitting auxiliary

[1] Multiple concurrent frontrunning attack campaigns fall outside our main threat model; see additional discussion in Appendix B.



(a) Transaction receive times at each replica

(b) Condorcet cycle and possible paths

Figure 2: A toy example of the Ambush attack.

transactions, which can be refunded if the attack is aborted in the middle; see more details in Section 8.1.

The adversary replica does *not* have any special privileges in the system. It cannot control or learn the transaction queues of other replicas, and it cannot manipulate or predict the behavior of other replicas. The adversary also does not have any special network privileges. It cannot manipulate the network latency of other replicas (e.g., delaying reception of a user transaction at other replicas), and it cannot predict the network latency between the replicas. It also does not have exclusive, faster communication channels than other replicas or users.

4 FRONTRUNNING VULNERABILITY

This section introduces a new frontrunning attack, which we call the Ambush attack, against the practical batch-order fair system, Themis [30]. We focus on the attack's high-level intuition and feasibility in this section.

4.1 A Toy Example of the Ambush Attack

We overview the high-level idea of the Ambush attack with a toy example of a frontrunning attack in Figure 2, which we will use as a running example throughout the paper. In this example, the adversary controls Replica 1 among the total $n = 5$ replicas in the system. Figure 2(a) visualizes how the transactions t^*, t^F, and t^a are received at each replica. The adversary wishes to frontrun a target transaction t^* submitted by a regular user with his/her frontrunning transaction t^F. For simplicity, we assume for now that there exist only three transactions in the epoch: t^*, t^F, and t^a, but no other transactions from benign users; see Section 5.3 where we discuss and evaluate the cases with thousands of benign user transactions.

The adversary first creates a transaction t^a *even before* observing the target transaction t^* and submits it to Replica 2 and Replica 3, which we say the adversary *ambushes* the transaction t^a to these replicas. When the target transaction t^* appears in the system, all the replicas receive the target transaction t^* roughly at the same time because of the broadcast-submission model. This triggers the adversary to create and submit the frontrunning transaction t^F to all the replicas. Moreover, the adversary submits the transaction t^a, which was ambushed earlier to Replica 2 and Replica 3, now to Replica 4 and Replica 5, achieving the broadcast submission of t^a but with some time delay. At the end of the epoch, all replicas report their transaction sequences in the order they received them to the leader. Replica 1 also reports its transaction sequence to the leader (whose identity is out of interest here) in the following order: (t^F, t^a, t^*), which is a lie made by the adversary who controls Replica 1.

Figure 2(b) shows how the three transactions are interpreted by the leader replica. Following the state-of-the-art batched fair-ordering algorithms, the leader replica constructs a Condorcet cycle: $t^* \to t^F \to t^a \to t^*$. Unspooling this cycle, three orderings are possible: $t^a \to t^* \to t^F$, $t^F \to t^a \to t^*$, and $t^* \to t^F \to t^a$. When any of these orderings can be chosen by the leader replica, the adversary successfully frontruns the target transaction t^* with t^F only if the second ordering is chosen. The attack fails otherwise. Observing this simplified example, one might conclude that such attacks could easily be mitigated by performing a simple majority vote on the popular path (e.g., eliminating the possibility of selecting path 2); however, in scenarios involving more transactions than this toy example, such straightforward voting is not feasible.

Root cause: *temporary information asymmetry*. The Ambush attack exploits the temporary information asymmetry between two groups of non-overlapping replicas. One group learns about the ambush transaction t^a before observing the target transaction t^*, while the other group learns about it after observing t^*. This information asymmetry allows the adversary to create a Condorcet cycle with the target transaction t^* and the frontrunning transaction t^F. This information discrepancy is only temporary because all replicas eventually learn about all transactions before an epoch ends, but it is sufficient for the adversary to conduct a successful frontrunning attack. We keep the discussion of the root cause of the attack in this section brief (to rather focus on the high-level intuition of the Ambush attack) and provide a more detailed explanation in Section 6.

Note that, however, the possibility of causing such information asymmetry is *not* a flaw in the batch-order fairness definition [30, 31]; instead, it is due to the artifact of the optimized network model, which has been introduced to improve the system's performance and scalability. In this regard, the proof of the impossibility of frontrunning attacks in Themis [30] (see Section 6.4 in [30] and Lemma C.2 in the extended version [29]) is based on the assumption that all replicas receive user transactions roughly at the same time, which turns out to be violated in the Ambush attack.

4.2 Challenges in Practical Attacks

As we saw in the toy example, the adversary needs to send some transactions (e.g., t^F and t^a) to other replicas before and after observing the target transaction to successfully frontrun it. However, in practical fair-ordering systems, benign users submit their transactions to the replicas in the same epoch and thus the adversary may not expect the exact desired transaction sequences illustrated in Figure 2(a). This raises a key question: *how can we design effective frontrunning attacks that function reliably in practical settings where numerous benign user transactions coexist within the same epoch?*

Our approach: Attack as a Template. Our solution for executing the Ambush attack in practical settings is to design the attack as a *transaction template*. Specifically, the adversary prepares a small set of adversary-generated transactions and a single potential target transaction, which constitute the elements in the transaction template, and submits them to the system at pre-determined times, *disregarding* the presence of other user transactions appearing in the

same epoch. This approach greatly simplifies the attack's design and execution, as the adversary can proceed the prepared transactions according to the template *as though* no other user transactions exist in the same epoch.

This rather care-free execution of the attack is possible because the adversary can *simulate* the attack *in advance* with various transaction templates with arbitrary user transactions and varying environments to precompute the expected success rates of the attack. By testing multiple transaction templates, the adversary can identify the one that maximizes the average attack success rate. Using this optimized template, the adversary can then submit transactions accordingly and achieve favorable success rates in practice.

While the template-based approach simplifies the attack's execution, two critical questions remain:

- **Question 1:** *How to guarantee the trapping of a target transaction with the transaction templates?* In other words, how to design the transaction templates that always include the target transaction in the same cycle as the frontrunning transaction, even when many user transactions exist in the same epoch? This is a crucial question to answer to make the Ambush attack work in the first place. In fact, it is a relatively easier question to answer.

  Lemma 1 (Sufficient Condition for Trapping a Target Transaction). *A target transaction t^* is always enclosed in a Condorcet cycle with the frontrunning transaction t^F if the adversary submits the ambush transaction t^a to a half of the non-adversary replicas before observing t^* and submits it to the remaining replicas after observing t^* and broadcasting t^F.*

  Proof. Recall the three-transaction example in Figure 2. We can see that the submission of these two adversary-generated transactions (t^a and t^F) according to the plan in Figure 2(a) successfully creates a Condorcet cycle with t^* and t^F. Then, no matter how many user transactions are inserted between these three transactions, as long as their relative orders are preserved at each replica, the three conflicting pairwise orderings between t^*, t^F, and t^a are preserved. Thus, a Condorcet cycle is always created among t^* and t^F with the minimal size of three (see Figure 2(b)) or larger when some user transactions join the cycle. □

- **Question 2:** *How to optimize the attack success rate with the transaction templates?* Now that we know how to guarantee the trapping of a target transaction with the transaction templates, the next question is how to design the transaction templates that maximize the attack success rate. Our toy example in Figure 2 demonstrates only one out of three possible paths yield successful frontrunning but this should be improved in practice when many user transactions exist in the same epoch. This is a challenging question that deserves a detailed discussion in Section 5.1 and Section 5.2.

4.3 Walking Through End-to-End Attack Steps

We begin by walking through the end-to-end attack steps from the adversary's perspective. The very first step is to select a target system (e.g., a rollup chain, a sequencing service, or a layer-1 blockchain) that accepts transactions from users and offer some financial services. In this paper, we consider any target system that uses a batch-order fair system, such as Themis [30], as a target for the Ambush attack.

Reconnaissance: Select a target system. After selecting the target system and learning its system parameters (e.g., the number of replicas, the unspooling algorithm, and the epoch timing and duration), the adversary establishes criteria for the target transaction t^*; e.g., a swap transaction involving huge trade volume, a liquidation transaction, or any other transaction that the adversary wants to frontrun. The specific criteria used for selecting the target transaction are not the focus of this work and are left to the adversary's discretion. Readers interested in additional examples of transaction selection criteria can refer to recent studies such as [39, 43, 48].

The Ambush adversary can now proceed with the attack by following the steps illustrated in Figure 3: offline preparation, Step 1 (ambush some transactions), Step 2 (trap the target transaction), and the end of attack (wait for finalization).

Offline preparation: Create a transaction template. Our Ambush adversary creates a transaction template tailored to the target fair-ordering system, especially considering its unspooling algorithm, and the number of replicas. Let us define the terms related to the transaction template. $\widehat{T}$ is the set of transactions used in the transaction template; e.g., $\widehat{T} = \{t^a, t^F, t^*\}$ in the toy example. A transaction template $\widehat{\mathcal{S}}$ is a set of transaction sequences, $\widehat{\mathcal{S}} = \{\widehat{S}_1, \ldots, \widehat{S}_n\}$, where each sequence $\widehat{S}_i$ is a permutation of $\widehat{T}$; i.e., $\widehat{S}_i \in \text{Perm}(\widehat{T})$. Figure 3 shows an example of the transaction template $\widehat{\mathcal{S}}$ from the toy example. Note that the adversary generates all transactions in $\widehat{T}$, except for the target transaction t^*, which is submitted by any benign user.

Step 1: Ambush some transactions in advance. With the transaction template $\widehat{\mathcal{S}}$ prepared, the Ambush adversary submits some transactions (or ambush transactions) at the beginning of a new epoch. In the example in Figure 3, the adversary uses a user account under her control to submit the transaction t^a to Replica 2 and Replica 3 before the target transaction t^* is submitted. The adversary is advised to submit ambush transactions as early in the epoch as possible to maximize the chance that they are received by the replicas before t^*, whose submission time is not known in advance. Notice that other users may also be submitting transactions to the replicas during this period.

Between Step 1 and Step 2. The adversary then waits for the target transaction t^* to be submitted by a benign user. If such a target transaction t^* meeting the adversary's criteria (e.g., a large MEV transaction, a liquidation transaction) is observed during the epoch, the adversary decides to proceed to Step 2. Otherwise, the adversary can abort the attack by refraining from submitting the remaining transactions in the transaction template $\widehat{\mathcal{S}}$. Figure 3 illustrates this interim period. In the case of an attack abortion, the adversary is refunded the cost of creating the ambush transactions; see Section 8.1 for more details.

Step 2: Trap the target transaction. The adversary now uses another controlled user account to submit the remaining transactions from the transaction template $\widehat{\mathcal{S}}$ to the appropriate replicas. In Figure 3, the adversary submits the frontrunning transaction t^F to all replicas and sends the ambush transaction t^a to Replica 4 and

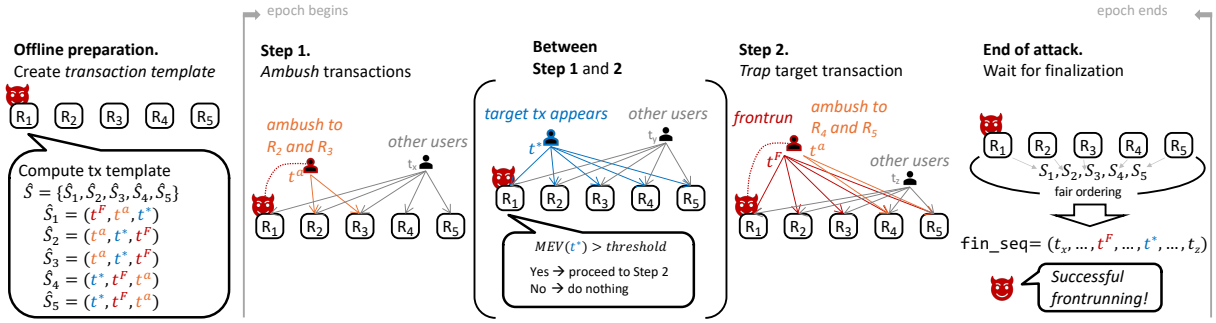


Figure 3: The AMBUSH attack steps.

Replica 5. When submitting multiple transactions to a replica, the adversary must ensure that they are submitted in the exact order specified by the template and before the current epoch ends.

End of attack: Wait for finalization. The adversary then waits for the epoch's final sequence (`fin_seq`). If t^F appears before t^* in the final sequence, the adversary has successfully frontrun the target transaction t^*. Otherwise, the attack fails.

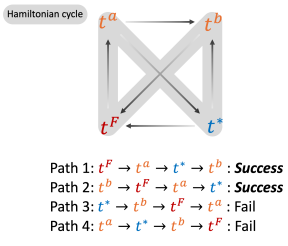
Monetization and post-mortem analysis. After viewing the final sequence, the adversary completes the monetization and analyzes the attack results. If the frontrunning is successful, the adversary may proceed with further steps to complete the MEV exploitation; for example, submitting a backrunning transaction to carry out a full sandwich attack. Once the attack is concluded, the adversary can evaluate the overall effectiveness of the campaign by comparing the direct profit (e.g., from arbitrage or liquidation) with the cost of the attack (e.g., creating and submitting ambush transactions); refer to Section 8.1 for further discussion of attack cost and profit.

5 COMPREHENSIVE EVALUATION OF THE AMBUSH ATTACKS

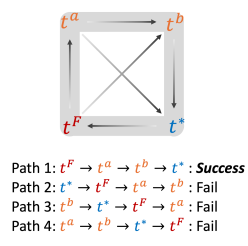
This section provides the intuitions for offline template optimization and the unknown factors that can be tested during the offline optimization. We then comprehensively evaluate the optimized attacks under various realistic settings.

5.1 Intuitions for Possible Optimization

Let us first offer some high-level intuitions of why the transaction templates can be optimized to achieve higher success rates in the frontrunning attack. We provide two different intuitions: (1) the graph structure, and (2) the number of transactions in a cycle.



(a) Graph with a high success rate



(b) Graph with a low success rate

Figure 4: Two cycles with different edge directions resulting in different success rates of frontrunning.

Intuition 1: Edge directions matter. The first reason is that the edge directions in the graph can create more favorable conditions for frontrunning during the unspooling process. Let us look at the two examples Condorcet cycles in Figure 4. The first graph, Figure 4(a), shows a frontrunning favorable graph. This is because if the unspooling algorithm that looks for a Hamiltonian path selects t^F or t^b as the starting vertex, t^F always precedes t^*, thus leading to a successful frontrunning; see possible permutation of the Hamiltonian cycle in Figure 4(a). In contrast, the second graph, Figure 4(b), shows a less favorable graph for frontrunning because the frontrunning is only possible if t^F is the starting vertex. This simple example shows that the edge directions in the graph of a Condorcet cycle can significantly affect the success rate of the frontrunning attack.

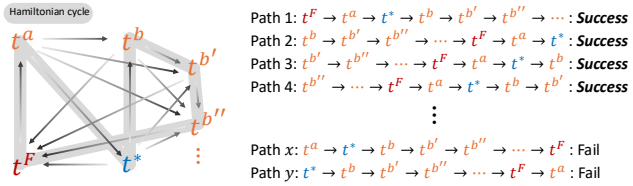


Figure 5: Adding more transactions with careful edge directions can increase the success rate of frontrunning.

Intuition 2: Larger cycles might help. Another dimension to increase the attack success rate is to add more transactions into a cycle. The edges of the added transactions should be carefully designed so that Hamiltonian paths that include these transactions would more likely traverse the frontrunning transaction before the target transaction. Figure 5 shows an example of a Condorcet cycle that is generated by adding more transactions to the cycle in Figure 4(a). We illustrate $t^{b'}$ and $t^{b''}$ as the added transactions for simplicity but in practice, many more transactions can be added. When unspooling the graph, the frontrunning transaction t^F is more likely to precede the target transaction t^* because the Hamiltonian path that traverses $t^{b'}$ or $t^{b''}$ first must pass t^F before reaching t^*. Therefore, adding more such transactions to the cycle can potentially increase the success rate of the frontrunning attack.

5.2 Offline Optimization

We now discuss how the adversary can optimize the transaction templates offline to increase the success rate of the frontrunning

attack. This means that the adversary can pre-compute the transaction templates in advance and use them in the actual attack in real-time. The crux of the idea is to test as many unknown factors as possible in the offline phase with local simulations and then craft the transaction templates that are most likely to succeed in the actual attack.

We summarize three main types of unknown factors that can be tested in the offline phase: (1) ordering of local sequences, (2) quorum selection, and (3) background benign transactions. We apply different testing strategies for each type of unknown factor.

Optimization 1: Pre-computation for ordering of local sequences. Let us first assume that there exists only one target transaction t^* and adversary-submitted transactions in the epoch, for simplicity. The leader replica after receiving all S_i's runs $\texttt{FairOrdering}(\mathsf{S}_1, \cdots, \mathsf{S}_n)$ (see Figure 3), which is a deterministic algorithm. The function $\texttt{FairOrdering}(\mathsf{S}_1, \cdots, \mathsf{S}_n)$ is deterministic and all S_i's are known to the adversary (because he/she created them); however, the adversary does *not* know the order of reception of S_i's by the leader replica because it is out of the adversary's control. The order of reception of S_i's affects the final order of transactions in the epoch as it is used as a part of the input to the deterministic algorithm `FairOrdering`.

In order to compensate for the unknown order of reception of S_i's, the adversary can simulate *all* possible scenarios of the order of reception of S_i's and simulate the unspooling process for each scenario, thus creating one $\widehat{S}$ transaction template (or several ties) that performs the best in the average case.

Optimization 2: Pre-computation for unknown quorum selection. In the practical batch-order fair systems, the final sequence is determined by the quorum of $(n - f)$ local sequences, where f is the maximum number of faulty replicas. The adversary does *not* know in advance which $(n - f)$ local sequences will be included in the quorum. In the similar manner as the unknown order of reception of S_i's, the adversary can simulate all possible combinations of the quorum of $(n - f)$ local sequences to find the best transaction template. Yet, there exists one additional complication: the adversary must ensure that the target transaction t^* and his/her frontrunning transaction t^F are trapped in the same Condorcet cycle in all possible combinations of the quorum of $(n - f)$ local sequences. This requires the attacker to submit additional ambush transactions to create a Condorcet cycle in all possible combinations of the quorum. Due to the space limitation, we provide a more detailed explanation in Appendix A.

Optimization 3: Pre-computation for background benign transactions. In practice, there can be many benign transactions in the epoch that are neither controlled nor known in advance by the adversary. Since it is computationally infeasible to simulate all possible cases involving background benign transactions, the adversary can take a best-effort approach to find better transaction templates. That is, the adversary can test a number of potentially high-performing transaction templates by conducting several large-scale simulations with different background benign transactions. The adversary can then select the transaction template that performs the best on average.

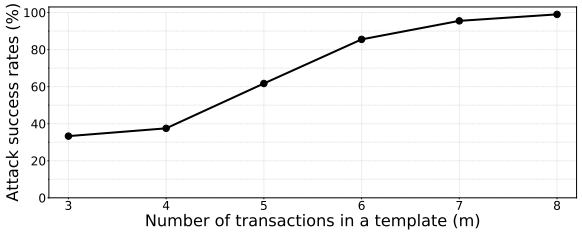


Figure 6: Attack success rates against Themis' unspooling algorithm for varying numbers of transactions in a template.

5.3 Evaluations with Optimized Templates

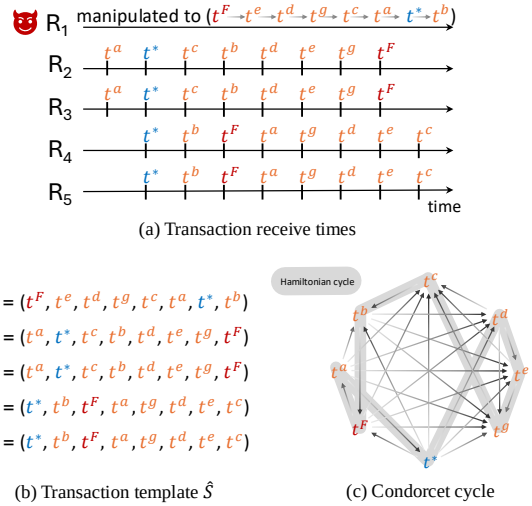
We evaluate the success rate of the frontrunning attack with the transaction templates, that are optimized with **Optimization 1, 2,** and **3**.

5.3.1 Evaluating Ambush with No Benign Transactions. We apply the **Optimization 1** strategy to find the best transaction templates, assuming that no benign transactions exist in the epoch. Since there exists one target transaction t^* and adversary-submitted transactions in the epoch, we can simulate all possible scenarios of the order of reception of S_i's in a reasonable amount of time.

The evaluation setup is simple: we create a set of m transactions that include the target transaction t^*, the frontrunning transaction t^F, and $m - 2$ additional auxiliary transactions (some of which are ambush transactions to create a Condorcet cycle). Then, we submit the set of transactions to the unspooling algorithm of Themis [30] (we use the open-source version [4]). This way, we can simulate the unspooling process for all possible scenarios of the order of reception of S_i's against Themis.

Figure 6 shows the best average success rates of the optimized transaction templates against Themis for varying numbers of transactions m. The success rate begins at 33.3% for $m = 3$ (as we saw in the toy example in Figure 2) and keeps increasing as m increases, until it reaches 99% for $m = 8$, where we stop the search. This result confirms our Intuition 2 in Section 5.1 that the larger the template size m, the higher the chance of success in the frontrunning attack can be expected. We also test the Ambush attack against unspooling algorithms other than Themis, and we leave the detailed results in Appendix D.

To better understand the high success rate of the optimized transaction templates, we present an example template and the resulting Condorcet cycle against Themis for $m = 8$ in Figure 7. This figure shows a slightly more complex example than our earlier toy example. Note that we maintain the similar style and color scheme in these figures to help readers understand the structure of the transaction templates and the resulting Condorcet cycles. Figure 7(a) depicts one particular example of how an adversary that controls Replica 1 submits transactions t^a, t^b, t^c, t^d, t^e, and t^g help create a Condorcet cycle. When the transactions are received as shown in Figure 7(a), the five local sequences shown in Figure 7(b) emerge, as expected by the adversary. The local sequences shown in Figure 7 are equivalent to the adversary's intended transaction templates (because we assume no benign transactions at the moment). These local sequences are sent to the leader node, where a Condorcet cycle of size 8 is formed as shown in Figure 7(c). This cycle is unspooled



(a) Transaction receive times

(b) Transaction template $\hat{S}$

(c) Condorcet cycle

Figure 7: Example transaction template and the resulting Condorcet cycle against Themis for $m = 8$.

by the Themis algorithm, which finds the one and only Hamiltonian cycle on the graph first, as shown as the shaded path in Figure 7(c). The one and only Hamiltonian cycle can then be rotated to create a Hamiltonian path (which becomes the final sequence `fin_seq`). We find empirically that all paths on the cycle (except the ones starting at t^a and t^*) results in a 99% frontrunning attack success rate.

5.3.2 Evaluating Attacks with Many Benign Transactions. We now apply **Optimization 2** and **3** with the presence of many benign transactions in the epoch. We consider realistic scenarios where n replicas are distributed across the globe. We evaluated the attack under both simulation and real-world settings. For simulations, we utilize the WonderProxy dataset [51] that provides the round-trip time (RTT) measurements between 250 globally distributed servers with their city name tags. Additionally, we measure the latency between servers in 21 different regions 1,500 times using Amazon Web Services (AWS) EC2 instances. The average RTT values for the two datasets are approximately 147.47 ms (standard deviation: 90.84 ms) and 139.87 ms (standard deviation: 87.06 ms), respectively. For the real-world deployment, we utilize 36 AWS m5.xlarge EC2 instances in 29 regions to run users and replicas. For both simulation and real-world deployment, all replicas and users are distributed across different regions. Our simulation code can be found at [5, 6] for reproducibility. We vary the number of benign user transactions in each epoch (which lasts for 1 second in all our simulations) from 0 to 1,500. These user transactions are generated at random times following the Poisson process, with the mean value adjusted to the total number of transactions in the epoch.

Figure 8 shows how the adversary can select the best-performing transaction templates in the presence of many benign user transactions. In this example, the adversary chooses four best-performing transaction templates from the previous optimization results. The attack success rates are then evaluated against Themis by varying the number of benign user transactions from 0 to 1,500. Note that

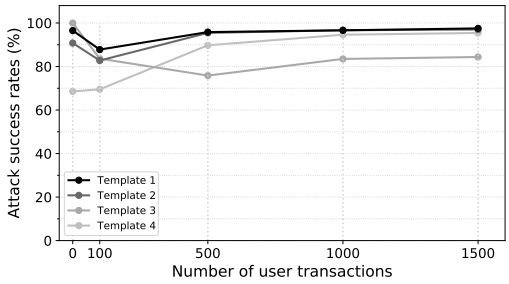


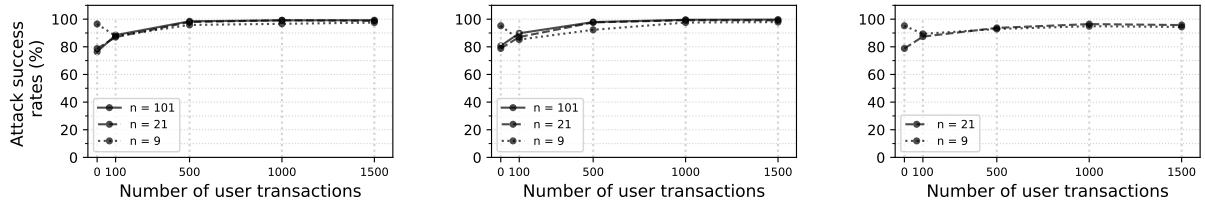
Figure 8: Attack success rates for several best-performing templates as the number of user transactions increases.

these chosen templates are optimized for both the order of local sequences and the unknown quorum selection. The adversary can select one template that outperforms the others at the expected number of user transactions in the epoch.

Let us first present the most comprehensive evaluation results of the Ambush attack in Figure 9, where we test the best templates against Themis with three system sizes, 9, 21, and 101 replicas (for real-world deployment, we only test with system sizes of 9 and 21 for practical reasons), to see whether the attack success rates are affected by the system size. For each data point, we created 50 evaluation instances, using different geographic locations from the dataset to simulate the users and the replicas at random. We executed the `FairOrdering` algorithm with 1,000 distinct reception orders of local sequences and 100 different quorum combinations for each instance. For all these cases, the attack success rates are calculated by counting the number of times frontrunning succeeded in the resulting final sequences.

Figure 9(a), (b), and (c) show the simulation results with our own AWS (time-varying) RTT measurements, the WonderProxy dataset, and the real-world deployment, respectively. All three results show similar trends and values, indicating that the simulations faithfully reflect the real-world network latencies, and that the attack success rates are not significantly affected by the network latencies. Also, the attack success rates are not significantly affected by the system size (i.e., $n = 9$, 21, or 101), suggesting that the attack depends less on the size of the system. Attack success rates against Themis are around 80% to 90% when 0 user transactions are present, but they increase to 90% to near 100% as the number of user transactions goes beyond 500.

Why do more user transactions result in higher success rates? It appears that increasing the number of user transactions helps the attack against Themis. While this aligns with Intuition 2 in Section 5.1 (i.e., more transactions in a cycle might help the attack), the exact reason is more nuanced. Let us investigate more. The main reason behind this improvement is that Themis constructs a Hamiltonian *cycle first* to find a Hamiltonian path later in the Condorcet cycle. Typically, a Condorcet cycle may have many Hamiltonian paths but only a few Hamiltonian cycles. The chosen transaction templates against Themis typically lead to *only one* Hamiltonian cycle in the Condorcet cycle so that the adversary can select a path only from the cycle. This two-step approach taken by Themis is a critical design choice that greatly simplifies the Hamiltonian path



(a) Simulation with AWS dataset (b) Simulation with WonderProxy dataset (c) Real-world deployment

Figure 9: Comprehensive attack simulations and real-world experiment against Themis with varying network sizes.

search problem, making Themis scalable to thousands of transactions per epoch. When there exists one Hamiltonian cycle in the Condorcet cycle, a user transaction added to the Condorcet cycle is much more likely to extend the existing Hamiltonian cycle rather than creating a new Hamiltonian cycle that degrades the attack; see our extended discussion in Appendix E.

Distance between frontrunning and target transactions. A frontrunning attack is considered successful if the frontrunning transaction is placed before the target in the final sequence. This holds true in many frontrunning attacks; for example, the profits of frontrunning liquidations are unaffected by the distance, since the adversary profits simply by placing the frontrunning transaction ahead of the target. Yet, sandwich attacks' profits may be influenced slightly by this distance (as discussed by Qin et al. [43]), since the asset token price could fluctuate due to intervening transactions. In our analysis of final sequences with 1,000 benign transactions inserted, we observe approximately 400 transactions, on average, positioned between the frontrunning and target transactions.

6 UNDESIRABLE TRADE-OFFS

Understanding the effectiveness of the Ambush attack, this section concerns the trade-offs between security and performance in the practical batch-order fair systems [30]. We first extend our discussion on the root cause of the vulnerability we had briefly in Section 4.1 and discuss ways to remove the root cause. We then provide empirical evidence that shows the undesirable trade-offs around frontrunning attacks.

6.1 Root Cause Analysis

As we briefly discussed earlier in Section 4.1, the root cause of the presented frontrunning vulnerability lies in a specific condition called *temporary information asymmetry* between two non-overlapping sets of replicas. Information asymmetry is, to some extent, an inherent characteristic of any distributed system. When new information is generated and disseminated, there is inevitably a delay before all replicas receive it. During this delay, information asymmetry arises temporarily, as some replicas gain access to the new information earlier than others.

In the context of our Ambush attack, we are particularly concerned with cases where this information asymmetry persists for a sufficiently long duration. Specifically, a distributed system is said to exhibit temporary information asymmetry if the time gap between the first and second sets of replicas receiving the new information is large enough to be exploited for the Ambush attack. The time gap is typically some fraction of the epoch length (e.g., from a few tens of milliseconds to a few seconds) because the Ambush attack attempt succeeds when the asymmetry persists even after the appearance of a target user transaction.

In the practical batch-order fair systems, such as Themis [30], the network model is leader-based, where the leader collects the local sequences from all replicas. Each replica is expected to have its own, independent observation of user transactions. This so-called *broadcast-submission model* not only ensures the independent observation of user transactions but also improves the system's performance and scalability as it avoids the need for any broadcast communication among replicas. However, the broadcast-submission model also introduces the possibility of temporary information asymmetry, unfortunately, because replicas do not broadcast received transactions to other replicas.

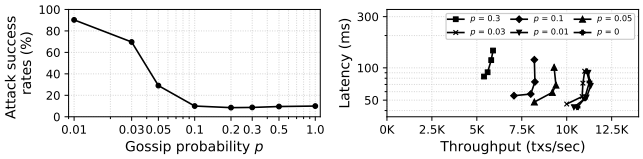
6.2 How to Remove the Root Cause?

When decentralized replicas experience temporary information asymmetry, the system is said to be in a state of temporary partitioning with respect to some transaction information; that is, the amount of information within one partition of replicas is different from that in another partition. One way to break this temporary partitioning (thus removing the root cause of the Ambush attack) is to ensure that all (or at least a majority of) replicas receive any new information quickly enough so that any temporary information asymmetry does not persist for a long time.

This, in fact, can be achieved by implementing some information dissemination mechanisms among replicas. However, as discussed above, the very reason for the practical batch-order fair systems to adopt the broadcast-submission model is to improve the system's performance and scalability. Therefore, implementing information dissemination mechanisms may come at the cost of performance degradation, which makes an undesirable trade-off.

6.3 Trade-offs: Testing via Partial Gossiping

Now to understand the trade-offs between frontrunning attack resistance and system performance in the practical batch-order fair systems, we test the Themis system [30] with a simple information dissemination mechanism that can be controlled by a single system parameter. This, what we call a partial gossiping mechanism, allows a replica to gossip about a newly observed user transaction with a probability p to another replica. To be specific, upon reception of a new transaction, a replica determines for each other replica whether to gossip about the transaction with probability p. Thus, when p is set to 1, each replica gossips all new user transactions to all other replicas, which is equivalent to the full broadcast primitive. When



(a) Attack success rates for varying gossip probability p

(b) Throughput vs. latency for varying gossip probability p

Figure 10: Attack success rates and system performance for varying gossip probability p.

p is set to 0, the replica does not gossip at all, which is equivalent to the current broadcast-submission model.

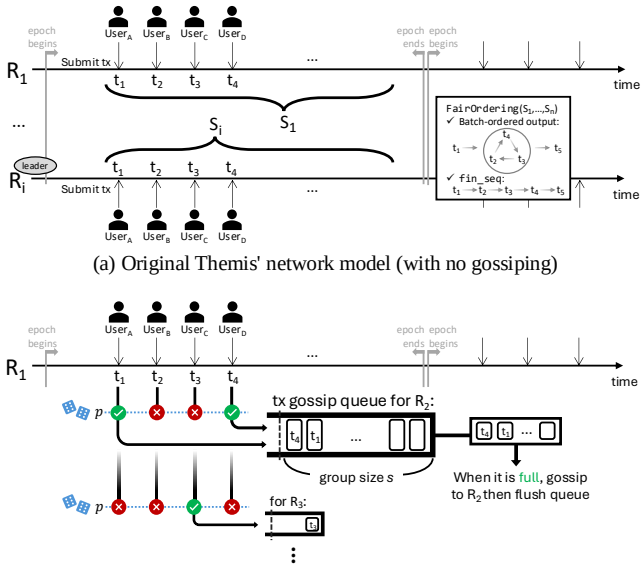
Figure 10 shows the undesirable trade-offs between the attack success rate and system throughput performance for varying the amount of information dissemination among replicas in the Themis system. Figure 10(a) shows the attack success rate for varying gossip probability p in the simulation environment. We re-use the simple attack simulation tool used in Section 5.3 to test the partial gossiping mechanism with varying probability p. It clearly shows that the attack success rate decreases as the gossip probability p increases, reaching nearly minimum when p is set to about 0.1. While this is an encouraging result, it also shows that the network should gossip with a non-negligible probability to prevent the Ambush attack successfully.

Figure 10(b) shows, unfortunately, that the system throughput decreases rather significantly as the gossip probability p increases. For this performance evaluation, we conduct a real-world multi-replica evaluation with the open-source Themis implementation in C++ [4]. The throughput and confirmation latency are measured on five servers with 110 MByte/s of network bandwidth, 21 replicas, and a block size of 100. To ensure that our performance evaluation does not incur any unnecessary overhead, we only utilize the existing long-lived replica-to-replica connections that are already established for the underlying consensus protocol. Specifically, these connections come from the same network library (i.e., salticidae [53]), which manages all consensus-related messages (e.g., proposals, votes). Over these connections, we exchange user transaction messages that are triggered by the partial gossiping mechanism. Compared to the baseline performance with $p = 0$, the throughput drops to about 65% when p is set to 0.1. This performance degradation is due to the additional communication overhead incurred by the partial gossiping mechanism.

These empirical results suggest that it is a non-trivial task to balance the trade-offs between the attack resistance and system performance when designing any secure and efficient information dissemination mechanism among replicas in the practical batch-order fair systems.

7 PRACTICAL MITIGATION

Understanding and measuring the trade-offs between security and performance in the network model in the practical batch-order fair system, we now investigate a practical countermeasure to effectively mitigate the Ambush attacks with minimal performance degradation.



(a) Original Themis' network model (with no gossiping)

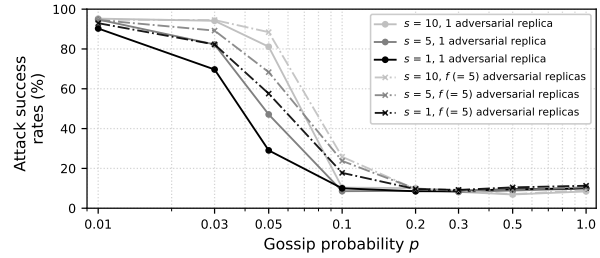
(b) Themis with the proposed delayed gossiping mechanism

Figure 11: A replica-level illustration of how the proposed delayed gossiping mechanism works in detail.

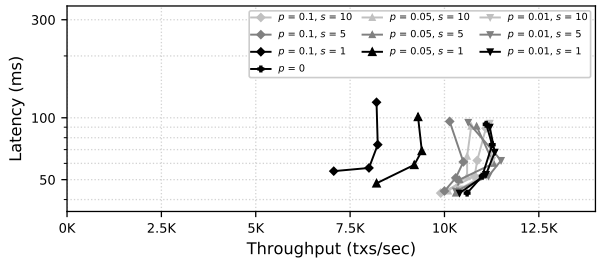
The key to finding such a balance is to understand how much temporal information asymmetry is *tolerable* for the system to prevent the Ambush attack successfully. To be specific, when there exists a temporary asymmetry between two sets of replicas due to the Ambush attack's certain transactions, the asymmetry should be broken but not necessarily immediately. The asymmetry can be broken after some time, as long as it is before the reception of the target transaction t^* by all replicas. As long as the adversary-generated ambush transactions are propagated to all replicas before the target transaction t^* arrives at all replicas, the Ambush attack cannot trap the target transaction t^* in a Condorcet cycle with the frontrunning transaction t^F.

This observation implies that the dissemination of transactions may get delayed at times while successfully mitigating the Ambush attack, opening the door to more efficient gossip strategies. To be more specific, replicas can delay the information dissemination while accumulating many transactions in a single gossip message. Gossiping this grouped transactions to another replica for breaking information asymmetry potentially reduces communication overhead.

Ideally, the system may want to gossip the grouped transactions right before a target transaction appears so that it can accumulate as many transactions as possible; however, assuming the knowledge of target transaction reception time is impractical. Instead, the system can set the target maximum delay for transactions and gossip grouped transactions frequently enough to break asymmetry before a target transaction appears. To avoid any additional complexity, we can tune the group size to set the approximate target delay that should be suitable for the certain operation setting of the system. For example, in a system with a 1-second epoch length, we may want to set the target delay to 0.1 seconds so that a replica gossips to another replica every 0.1 seconds on average.



(a) Attack success rates with different group sizes



(b) Throughput vs. latency with different group sizes

Figure 12: The effectiveness of the delayed gossiping mechanism for varying gossip probability p and group size s.

Given an expected transaction-per-second (TPS) performance of the system and gossip probability p, we can estimate the number of accumulated transactions to be gossiped in 0.1 seconds. By setting the group size to the estimated number, we can make the replicas frequently gossip with the target delays on average.

To that end, we propose a simple yet effective countermeasure called *delayed gossiping* which uses two security parameters, gossip probability p and group size s, for tuning the effectiveness and the overhead of gossiping. Figure 11(a) shows the unmodified version of Themis, where users broadcast to submit their transactions to all replicas and each replica independently receives transactions without gossiping. Figure 11(b) illustrates an example of how a replica gossips a group of transactions using delayed gossiping. When a replica receives a new transaction, the replica decides whether to gossip the transaction based on a gossip probability p. If the replica decides to gossip the transaction, it is inserted into the transaction gossip queue. Once the queue length reaches the group size s, the replica groups all the transactions into a single gossip message, sends it to the corresponding peer replica, and then flushes the queue. Notice that the replica maintains a distinct transaction gossip queue for every other peer.

Figure 12 shows the effectiveness of the delayed gossiping countermeasure for varying gossip probabilities and group sizes. We consider the system size of 21 replicas. As Figure 12(a) shows, as we increase the group size s from 1 to 5 to 10, the gossiping becomes less effective in mitigating frontrunning attacks (i.e., the attack success rate increases). This is because the larger group size delays the information dissemination, which can allow the adversary to maintain the temporary information asymmetry for a sufficiently long time for the Ambush attack to succeed.

While all our evaluations so far consider the adversary using a single replica, we also consider the case where the adversary can leverage f replicas to launch the Ambush attack. The adversary can use these multiple replicas to silently refuse gossiping transactions, thus degrading the mitigation's effectiveness and increasing the attack success rates. Our evaluation shows that the attack success rate is slightly higher when p is small, but as p increases, transactions are gossiped more frequently, eventually breaking the information asymmetry and reducing the attack success rates; see the dashdotted lines in Figure 12(a).

Figure 12(b) shows that the system throughput performance does not degrade significantly as the group size increases. As the group size s increases to 5, the partial gossiping with $p = 0.1$ demonstrates only about 10% performance degradation compared to the baseline performance with $p = 0$. When s increases to 10, the performance degradation becomes negligible. This is because the delayed gossiping significantly reduces the communication overhead, which can help maintain the system's performance and scalability.

Our empirical results in Figure 12 suggest that it is indeed possible to make a practical balance between security and performance, especially when we understand enough temporal information asymmetry we can allow in the system, which can be determined dynamically based on the system's configuration, the network model, and how many transactions are observed by all replicas.

8 DISCUSSION

We discuss several aspects of the presented Ambush attacks that are important for understanding the attack's economic feasibility, sustainability, and comparison to other attacks.

8.1 Attack Cost

The cost of the Ambush attack consists of two parts: (1) offline template computation overhead and (2) transaction fees. First, the template computation overhead is completely offline because it is done only once before the attack campaign. Therefore, it does not add any delay to the real-time frontrunning attacks. Also, template generation takes only a reasonable amount of time; for instance, generating a template for a 101-replica system takes about 150 minutes on a 48-core machine. This allows the adversary to compute multiple templates in advance and select the one with the highest success rate.

Second, the transaction fees for the Ambush attack are relatively low compared to the potential gains from frontrunning the target transaction. The Ambush attack requires to create and submit a few ambush transactions to the target fair ordering system and that is the only cost the adversary has to pay. When calculating the cost of the Ambush attack, we *exclude* the cost of fabricating the frontrunning transaction t^F from the cost of the Ambush attack because it directly depends on the value of the target transaction t^* and the entire attack campaign (e.g., sandwich attack) that the adversary is planning to execute. We also exclude the transaction values in the ambush transactions because the adversary can send some tokens to his/her other accounts. Thus, the cost of the Ambush attack is summarized as the total cost of the fees for the ambush transactions and the frontrunning transaction. With 101 replicas,

the attack against Themis uses 157 such transactions. Considering that transaction fees are about USD 0.005 in Arbitrum (as of July 2024 [38]), one of the possible target systems for fair-ordering systems, the cost of the Ambush attack ranges from USD 0.785 to USD 1.045 per attack attempt. This cost should be paid by the adversary regardless of the success of the attack.

One critical advantage of the Ambush attack is that the adversary can *recover* the attack cost in the epoch, if the adversary decides to abort the attack before sending the frontrunning transaction and the ambush transactions to the rest of the replicas. That is, if the adversary decides to stop the attack after Step 1 (see Figure 3), the ambush transactions that have been sent to some replicas are *not* included in the final sequence of the epoch, thus costing nothing to the adversary. This is because a transaction is inserted into the final sequence only when it appears more than $2f + 1$ replicas' local sequences; see Themis [30] for details.

8.2 Attack Sustainability

While the presented Ambush attack is not completely stealthy, it is sustainable because the adversary can keep utilizing disposable resources without its true identity being revealed. The Ambush attack may create unusual transaction patterns (e.g., ambush transactions) in the fair-ordering system and thus can be detected by a vigilant system operator. Yet, the Ambush adversary can continue the attack even after or during the active detection because the amount of resources required for the attack is low and they are disposable. The adversary makes his/her own user account(s) to send some ambush transactions and the frontrunning transaction. In blockchains that accept transactions from users accounts without any permission (e.g., Ethereum, Arbitrum, etc.), it is hard to relate these accounts to the true identity of the adversary behind. A cautious adversary can further hide his/her identity by utilizing multiple network identities (e.g., IP addresses) and multiple user accounts, which incur additional costs but it is still affordable. Moreover, the ambush transactions themselves are not malicious; they are just benign-looking transactions that are sent to the adversary-chosen replicas at the adversary-chosen time.

8.3 Comparison to the Condorcet Attack

Recent work by Vafadar et al. [49], called the Condorcet attack, shares some similarities to the presented Ambush attack as both intentionally create Condorcet cycles in the fair-ordering systems. However, the two attacks are largely different and should not be confused with each other. The main attack goal of the Condorcet attack is degrading order fairness of the fair ordering system with a malicious client, and never considering frontrunning attacks. In contrast with the Condorcet attack, Ambush attack aims to frontrun a target transaction with a single malicious replica. One characteristic that may make the two attacks look similar is the strategy that both attacks create Condorcet cycles. Yet, their strategies in constructing the Condorcet cycle are different; Ambush attack is designed as a targeted attack and requires a precise design of the Condorcet cycle while the Condorcet attack is not. Furthermore, the Condorcet attack does not consider the quorum-based ordering for liveness, therefore, it does not always guarantee the construction of the Condorcet cycle when critical local sequences from replicas are missing, while the Ambush attack can always make the Condorcet cycle by using well-designed templates (Appendix A). In short, the Ambush attack is different from the Condorcet attack as it does not share the goals or the attack techniques. Furthermore, our work goes beyond the presentation of new attacks against existing practical batch-order fair systems by offering insights into the structural trade-offs between the security and performance of fair ordering systems. Table 1 summarizes the key differences between the Ambush attack and the Condorcet attack.

Table 1: Comparison between the Ambush vs. Condorcet attacks [49].

	Ambush attack	**Condorcet attack [49]**
Attack goal	Frontrun a target transaction	Degrade order fairness (frontrunning is never a goal)
Create a Condorcet cycle	Yes	Yes
Perpetrator	Single replica	Single client
Precise cycle design	Required	Not required
Targeted attack	Yes	No
Consider quorum-based ordering	Yes	No

9 CONCLUSION

In a world increasingly reliant on blockchain technology for diverse transactions, frontrunning remains a critical challenge that demands continuing attention. In this work, we demonstrate that a state-of-the-art first-come-first-served ordering system specifically designed to mitigate frontrunning is, unfortunately, entirely vulnerable to a novel form of attacks due to a subtle yet critical trade-off in its network design. While this specific vulnerability can be handled as we demonstrate, our findings highlight the need for a more careful understanding of the interplay between distributed algorithms and network protocols to better address this persistent issue.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers and our shepherd for their insightful comments and invaluable feedback. This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2025-RS-2023-00259099) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).

REFERENCES

[1] Mohammad Javad Amiri, Heena Nagda, Shubhendra Pal Singhal, and Boon Thau Loo. 2024. Rashnu: Data-Dependent Order-Fairness. In *Proc. VLDB*. 1–14.
[2] Dana Angluin and Leslie G Valiant. 1977. Fast probabilistic algorithms for Hamiltonian circuits and matchings. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. 30–41.
[3] Robert Annessi. 2022. Can First Come First Served-Based Transaction Ordering Prevent Front-Running? https://collective.flashbots.net/t/can-first-come-first-served-based-transaction-ordering-prevent-front-running/892/1. Accessed: 2025-01-02.

[4] Anon. 2022. Themis-code: Source Code for hotstuff with Themis. https://github.com/anonthemis/themis-src-anon.

[5] Anonynous authors. 2025. Ambush attack github repository. https://github.com/NetSP-KAIST/the-ambush-attack.

[6] Anonynous authors. 2025. Ambush attack zenodo repository. https://zenodo.org/records/15703035.

[7] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. 2018. A Fair Consensus Protocol for Transaction Ordering. In *Proc. ICNP*. 55–65.

[8] Kushal Babel, Nerla Jean-Louis, Yan Ji, Ujval Misra, Mahimna Kelkar, Kosala Yapa Mudiyanselage, Andrew Miller, and Ari Juels. 2024. PROF: Protected Order Flow in a Profit-Seeking World. *arXiv preprint arXiv:2408.02303* (2024).

[9] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. 2022. Sok: Mitigation of front-running in decentralized finance. In *Proc. FC*.

[10] Richard Bellman. 1962. Dynamic programming treatment of the travelling salesman problem. *J. ACM* 9, 1 (1962), 61–63.

[11] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2019. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1521–1538.

[12] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. 2018. Enter the hydra: Towards principled bug bounties and {Exploit-Resistant} smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1335–1352.

[13] Lorenz Breidenbach, Tyler Kell, Stephane Gosselin, and Shayan Eskandari. 2018. Defeat Front-Running on Ethereum. https://libsubmarine.org/. Accessed: 2024-12-29.

[14] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. 2022. Quick Order Fairness. In *Proc. FC*. 316–333.

[15] Wuhui Chen, Yikai Feng, Jianting Zhang, Zhongteng Cai, Hong-Ning Dai, and Zibin Zheng. 2024. Auncel: Fair Byzantine Consensus Protocol with High Performance. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*. 1251–1260.

[16] Jean Antoine Marie Nicolas Caritat Condorcet et al. 1785. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. De l'imprimerie Royale, Paris, France. (written in French).

[17] Andrei Constantinescu, Diana Ghinea, Lioba Heimbach, Zilin Wang, and Roger Wattenhofer. 2024. A Fair and Resilient Decentralized Clock Network for Transaction Ordering. In *Proc. OPODIS*. 8–1.

[18] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *Proc. IEEE S&P*. 910–927.

[19] NetworkX Developers. 2023. networkx.algorithms.tournament.hamiltonian_path — NetworkX 2.8.8 documentation. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.tournament.hamiltonian_path.html.

[20] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2019. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. In *Proc. WTSC*. 170–189.

[21] Ed Felten. 2022. L2 sequencing and MEV. In MEV.Day @Devconnect, Amsterdam, 2022, https://youtu.be/qxml80TparY?si=emHO84QNvIA3weB3 and https://docs.google.com/presentation/d/1wRlSiocBeg_FwwI7ScUnfQ8mil6dI1qzzePT--bx-I/edit#slide=id.g12566ee1ac5_0_116.

[22] Christof Ferreira Torres, Albin Mamuti, Ben Weintraub, Cristina Nita-Rotaru, and Shweta Shinde. 2024. Rolling in the Shadows: Analyzing the Extraction of MEV Across Layer-2 Rollups. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2591–2605.

[23] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM, USA. https://www.osti.gov/biblio/960616 and https://networkx.org/.

[24] Lioba Heimbach, Vabuk Pahari, and Eric Schertenleib. 2024. Non-Atomic Arbitrage in Decentralized Finance. In *Proc. IEEE S&P*. 1–18.

[25] Lioba Heimbach and Roger Wattenhofer. 2022. Sok: Preventing transaction reordering manipulations in decentralized finance. In *Proc. ACM AFT*.

[26] Michael Held and Richard M Karp. 1962. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math.* 10, 1 (1962), 196–210.

[27] Chi Ho, Danny Dolev, and Robbert Van Renesse. 2007. Making distributed applications robust. In *Principles of Distributed Systems: 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings 11*. Springer, 232–246.

[28] Ari Juels. 2020. Fair Sequencing Services: Enabling a Provably Fair DeFi Ecosystem. https://blog.chain.link/chainlink-fair-sequencing-services-enabling-a-provably-fair-defi-ecosystem/.

[29] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2021. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. Cryptology ePrint Archive, Paper 2021/1465.

[30] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2023. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. In *Proc. ACM CCS*. 475–489.

[31] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. In *Proc. CRYPTO*. 451–480.

[32] Klaus Kursawe. 2020. Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains. In *Proc. AFT*. 25–36.

[33] Offchain Labs. 2024. A gentle Introduction: Timeboost. https://docs.arbitrum.io/how-arbitrum-works/timeboost/gentle-introduction. Accessed: 2025-01-02.

[34] Offchain Labs. 2024. The Sequencer and Censorship Resistance. https://docs.arbitrum.io/how-arbitrum-works/sequencer. Accessed: 2024-12-28.

[35] Kai Li, Yuzhe Tang, Jiaqi Chen, Yibo Wang, and Xianghong Liu. 2021. TopoShot: Uncovering ethereum's network topology leveraging replacement transactions. In *Proceedings of the 21st ACM Internet Measurement Conference*. 302–319.

[36] Rujia Li, Xuanwei Hu, Qin Wang, Sisi Duan, and Qi Wang. 2023. Transaction Fairness in Blockchains, Revisited. Cryptology ePrint Archive, Paper 2023/1034.

[37] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. arXiv:2208.00940 [cs.CR]

[38] Marcov. 2024. Layer 2 - Transaction Cost. https://dune.com/Marcov/layer-2-transaction-cost.

[39] Robert McLaughlin, Christopher Kruegel, and Giovanni Vigna. 2023. A large scale study of the ethereum arbitrage ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3295–3312.

[40] Ke Mu, Bo Yin, Alia Asheralieva, and Xuetao Wei. 2024. Separation is Good: A Faster Order-Fairness Byzantine Consensus. In *Proc. NDSS*. 1–17.

[41] Rafael Pass, Elaine Shi, and Florian Tramer. 2017. Formal abstractions for attested execution secure processors. In *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part I 36*. Springer, 260–289.

[42] Kaihua Qin, Liyi Zhou, Pablo Gamito, Philipp Jovanovic, and Arthur Gervais. 2021. An empirical study of DeFi liquidations: incentives, risks, and instabilities. In *Proc. IMC*. 336–350.

[43] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *Proc. IEEE S&P*. 198–214.

[44] Frank Rubin. 1974. A Search Procedure for Hamilton Paths and Circuits. *J. ACM* 21, 4 (1974), 576–580.

[45] Chrysoula Stathakopoulou, Signe Rüsch, Marcus Brandenburger, and Marko Vukolić. 2021. Adding fairness to order: Preventing front-running attacks in bft protocols using tees. In *Proc. IEEE SRDS*. 34–45.

[46] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[47] Ana Tatabitovska, Oğuzhan Ersoy, and Zekiraya Erkin. 2021. Mitigation of transaction manipulation attacks in UniSwap.

[48] Christof Ferreira Torres, Ramiro Camino, and Radu State. 2021. Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain. In *Proc. USENIX Security*. 1343–1359.

[49] Mohammad Amin Vafadar and Majid Khabbazian. 2023. Condorcet Attack Against Fair Transaction Ordering. In *Proc. AFT*. 15:1–15:21.

[50] Ben Weintraub, Christof Ferreira Torres, Cristina Nita-Rotaru, and Radu State. 2022. A flash (bot) in the pan: measuring maximal extractable value in private pools. In *Proc. IMC*. 458–471.

[51] WonderProxy. 2020. A day in the life of the Internet. https://wonderproxy.com/blog/a-day-in-the-life-of-the-internet/.

[52] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proc. PODC*. 347–356.

[53] Ted Yin. 2021. Salticidae-code: Source Code for hotstuff with Themis. https://github.com/Determinant/salticidae.

[54] Pouriya Zarbafian and Vincent Gramoli. 2023. Lyra: Fast and Scalable Resilience to Reordering Attacks in Blockchains. In *Proc. IEEE IPDPS*. 929–939.

[55] Fengwei Zhang and Hongwei Zhang. 2016. SoK: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. 1–8.

[56] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *Proc. USENIX OSDI*. 633–649.

[57] Chonghe Zhao, Yipeng Zhou, Shengli Zhang, Taotao Wang, Quan Z Sheng, and Song Guo. 2024. DEthna: Accurate ethereum network topology discovery with marked transactions. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 1711–1720.

[58] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. 2021. On the Just-In-Time Discovery of Profit-Generating Transactions in DeFi Protocols. In *Proc. IEEE S&P*. 919–936.

A UNKNOWN QUORUM SELECTION

Themis uses $(n-f)$ local sequences to produce the final sequence fin_seq to ensure the liveness even with f faulty replicas. To this end, the adversary optimizes transaction templates by adding more ambush transactions so that it can always create a Condorcet cycle in *all* possible combinations of the quorum of $(n-f)$ local sequences. This indeed complicates the attack, as the adversary now needs to consider $\frac{n!}{f!(n-f)!}$ combinations of $(n-f)$ local sequences. Our Ambush attack should account for this quorum selection. The adversary must ambush $f+1$ different ambush transactions (not just a single ambush transaction as seen in Section 4) to create a Condorcet cycle in all possible combinations of the quorum of $(n-f)$ local sequences.

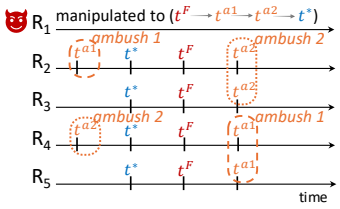


Figure 13: A toy example template for unknown quorum selection.

We explain how to optimize the templates with additional ambush transactions. Although the adversary does not know which local sequences will be excluded from the quorum, the adversary knows the number of local sequences (i.e., f) that will be excluded. Knowing this, the adversary can guess the local sequences that will be included in the quorum and create a template that only uses those local sequences to create a Condorcet cycle. If any of the guessed local sequences are not included in the production of the Condorcet cycle, the attack fails, and Condorcet cycle creation succeeds otherwise. To always reliably create Condorcet cycles, the adversary can overlap $f+1$ such guesses, so that even if at most f of the guesses are wrong, the adversary can still create one Condorcet cycle to conduct the attack. We use a simple example in Figure 13 to illustrate how we can send multiple ambush transactions to create a Condorcet cycle in all possible combinations of the quorum. In the example of $n=5$ (with $f=1$) replicas, the adversary should simulate total four cases of the quorum of $(n-f)=4$ local sequences: the set of all 3-element subsets of $\widehat{\mathcal{S}} \setminus \widehat{S}_1 = \{\widehat{S}_2, \widehat{S}_3, \widehat{S}_4, \widehat{S}_5\}$ or $\binom{\widehat{\mathcal{S}} \setminus \widehat{S}_1}{3}$ (since the replica 1 is the adversary-controlled replica). The example attack submits two ambush transactions t^{a1} and t^{a2} to create a Condorcet cycle in all the possible combinations of the quorum. Note that in the case when the adversary wishes to place more transactions into the cycle for the purpose of increasing the attack success rate (see Intuition 2 in Section 5.1), the adversary can still do so by submitting additional transactions *per* ambush transaction.

B CONCURRENT MULTIPLE ATTACK CAMPAIGNS

In all attack scenarios we discussed in the main body of the paper, we assume that the attack is conducted by a single adversary either with a single replica or multiple colluding replicas. While it is outside the main scope of this paper, we briefly discuss the implications of multiple concurrent attack campaigns on the same target transaction. For this discussion, we first extend the definition of attack success to include the attack success rate of multiple attackers: (1) the *single-winner* scenario and (2) the *multiple-winners* scenario. First, in the single-winner scenario, only one attacker succeeds, whose frontrunning transaction frontruns both the target and all the other frontrunning transactions. This applies to cases when only the first frontrunning transaction is profitable and the subsequent ones are not (e.g., liquidation). For this case, the attack success rate naturally tapers off as the number of attackers increases since they compete with each other to be the first among all.

Second, in the multiple-winners scenario, many attackers can succeed as long as their frontrunning transactions are placed before the target transaction. This scenario applies when multiple frontrunning transactions yield some profit (e.g., multiple overlapping sandwich attacks), although this is less common in practice. For this case, the average attack success rates decrease less dramatically compared to the previous scenario; for example, about $1\%p$ decrease of success rate when $f=2$, about $5\%p$ decrease when $f=5$, and about $20\%p$ decrease when $f=25$, according to our separate simulation results. This is because as more adversaries insert their ambush transactions, the complexity of the Condorcet cycle increases, deviating from their intentions. Nevertheless, the Condorcet cycle is still formed, enabling frontrunning to some extent.

C THEMIS' DETERMINISTIC UNSPOOLING ALGORITHM

In Themis [30], at the end of an epoch, the leader replica runs `FairOrdering()` — which first constructs a graph with transactions as in Figure 2, identifies strongly connected components (SCCs) using Tarjan's algorithm [46], and topologically sorts the SCCs — to produce the final sequence. If an SCC contains multiple transactions, the leader applies the deterministic cycle unspooling algorithm, which finds a Hamiltonian cycle[2], to the SCC, producing a sequence. Given an SCC with k transactions, the algorithm first constructs a path covering all k transactions, then identifies a subpath that forms a cycle, and finally swaps subpaths to form a Hamiltonian cycle. For example, given a path $t^1 \rightarrow \cdots \rightarrow t^i \rightarrow t^j \rightarrow \cdots \rightarrow t^k$, the algorithm finds a cycle-forming subpath $t^1 \rightarrow \cdots \rightarrow t^i \rightarrow t^1$, then merges the remaining paths into it: $t^1 \rightarrow \cdots \rightarrow t^j \rightarrow \cdots \rightarrow t^k \rightarrow \cdots \rightarrow t^i \rightarrow t^1$. After finding the cycle, a path is deterministically derived from the cycle and inserted into the final sequence. The full details are available in the open-source Themis repository [4].

D EFFECTIVENESS AGAINST VARIOUS UNSPOOLING ALGORITHMS

We conduct additional evaluations on the Ambush attack against various unspooling algorithms (including the one used in Themis). For a clear comparison, we conduct the evaluations without benign transactions. Some unspooling algorithms are representative algorithms used in the literature (e.g., backtracking [44], dynamic

[2]While not explicitly stated in the Themis paper or codes, the algorithm is a variant of Angluin and Valiant's algorithm [2].

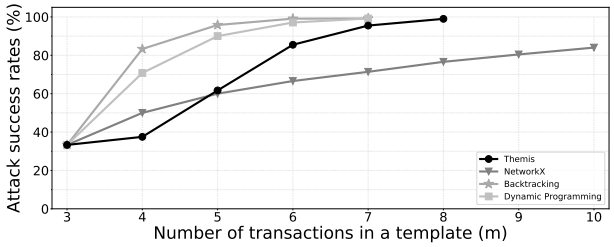


Figure 14: Attack success rates against various unspooling algorithms for varying numbers of transactions in a template.

programming [10]), and some are found in the popular open-source libraries (e.g., NetworkX [23]). All the unspooling algorithms we test are deterministic algorithms that respect the pairwise orderings of transactions in the graph; we do not consider algorithms that ignore the given graph structure (e.g., random walk algorithms without edge directions, alphabetical orderings).

The following are the unspooling algorithms we test:

- `NetworkX`: NetworkX [23] is a popular open-source Python library that contains various graph algorithms, including the Hamiltonian path algorithm [19]. The algorithm iteratively selects and orders vertices in a sequential manner, visiting each vertex exactly once. We use the unmodified algorithm with arbitrarily shuffled vertices as input.
- `Backtracking`: The backtracking algorithm [44] uses a depth-first search to find a Hamiltonian cycle. The algorithm traverses through the vertices based on the edge directions and constructs paths; once the path's length matches the total number of vertices, the algorithm determines if it can connect the path's start and end. If not, it moves back along the path to find a new path. It repeats this process until it finds a Hamiltonian cycle.
- `Dynamic Programming`: The dynamic programming method [10, 26] tries to find Hamiltonian paths by extending the found paths until the path includes all vertices in the graph. It uses arbitrarily shuffled vertices as input.

Figure 14 shows the attack success rates of the optimized transaction templates against these unspooling algorithms. As discussed earlier, the larger the template size m, the higher the chance of success in the frontrunning attack. When we achieve a maximum success rate of nearly 100% for certain unspooling algorithms, we stop the search for larger m. Thus, we stop the search at around m = 6 to 8 for all unspooling algorithms except `NetworkX` where we stop at m = 10 due to our computational resources.

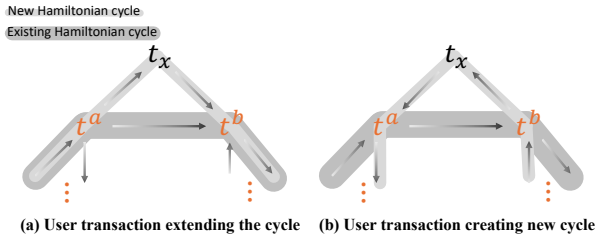
Let us discuss the evaluation results in more detail. First, for m = 3, the attack success rate is 33.3% regardless of the unspooling algorithm used, because they all use the same transaction template we saw in the toy example in Figure 2(a).

Attacks against `Dynamic Programming` and `Backtracking` show quickly increasing success rates as the number of transactions m increases, reaching 99% at $m = 7$ and $m = 6$, respectively. This is because although these unspooling algorithms choose a starting vertex (transaction) based on some unpredictable factors, they traverse the graph in a predictable manner to find a Hamiltonian path (in the case of `Dynamic Programming`) or a Hamiltonian cycle (in the case of `Backtracking`). For example, they extend the path by following the outgoing edge of the current path's last vertex and appending new vertices to the path until all vertices are included. Our results show that regardless of whether the unspooling algorithm seeks a Hamiltonian path or a cycle, the Ambush attack can effectively predict the path or cycle that places the frontrunning transaction before the target transaction, achieving very high success rates for sufficiently large m.

Attacks against `NetworkX` show a slower increase in success rate compared to the other algorithms for $m \leq 6$. The rate reaches 84.0% at $m = 10$, where we stop the search. Our analysis shows that the slow increase of success rate is mainly caused by how `NetworkX` differently constructs a Hamiltonian path compared to the other algorithms. While the other algorithms we test iteratively select the next vertex to extend the path only in the *forward* direction (i.e., from the starting vertex to the ending vertex), `NetworkX` extends the path in a more random manner, making the adversary's prediction of Hamiltonian paths more difficult. For example, `NetworkX` picks a vertex from the cycle, and it tries to even prepend or insert (in the middle) in addition to append to the current path until every vertex is included.

Overall, the evaluation results show that the Ambush attack is highly effective against the unspooling algorithms used in practice. The degree of effectiveness depends on the specific unspooling algorithm used but the attack can achieve high success rates against all the algorithms we test for large enough m.

E WHY MORE USER TRANSACTIONS INCREASE THE ATTACK SUCCESS RATE



(a) User transaction extending the cycle (b) User transaction creating new cycle

Figure 15: Why a new user transaction would likely increase the existing Hamiltonian cycle.

In our large-scale experiments in Section 5, we have shown that as the number of user transactions increases, the Ambush attack's effectiveness improves. Here, we argue that this is because when a new user transaction is added to the batch, it is much more likely to elongate the existing Hamiltonian cycle, rather than creating a new one. The extended cycle is then more likely to increase the attack success rate.

We first show that when a new user transaction is added to the batch of transactions, it is much more likely to *elongate* the existing Hamiltonian cycle. Figure 15 shows the two possible ways of altering the existing Hamiltonian cycle when a new user transaction is added to the batch. The existing cycle in the figure already includes the edge $t^a \rightarrow t^b$, indicating the majority of the nodes received t^a sooner than t^b. Figure 15(a) shows a case where the newly added user transaction t_x extends the cycle when its edge direction aligns with the ambush transactions' edge direction $t^a \rightarrow t^b$. As the majority of nodes received t^a then t^b, this extension may happen

naturally when t_x arrives at nodes between these two transactions, forming the relationship of $t^a \rightarrow t_x \rightarrow t^b$.

In contrast, it is unlikely for a benign user transaction to form a new Hamiltonian cycle shown in Figure 15(b). This is because, to form this new cycle, t_x's edges must be set in the opposite direction of the existing cycle, which is extremely unlikely to happen by chance. For example, to form the edge $t_x \rightarrow t^a$, the majority of nodes must receive t_x sooner than t^a; however, since the majority of the nodes received t^a sooner than t^b, the edge $t_x \rightarrow t^b$ is more likely to be created, failing at creating a new cycle shown in Figure 15(b). Likewise, creating this new cycle would require a similar amount of effort to conduct another Ambush attack, which is not expected from benign users.

When a new user transaction is added to the cycle, it is most likely to extend the existing cycle by making an ordering relationship of (a) with adjacent transactions. As the cycle size increases by elongating the existing cycle, the effect of Intuition 2 in Section 5.1 is strengthened, creating more frontrunning-succeeding paths and increasing the attack success rate.