DoubleUp Roll: Double-spending in Arbitrum by Rolling It Back

Zhiyuan Sun
The Hong Kong Polytechnic University
Hong Kong, China
Southern University of Science and Technology
Shen Zhen, China
zhi-yuan.sun@connect.polyu.hk

Zihao Li
The Hong Kong Polytechnic University
Hong Kong, China
cszhli@comp.polyu.edu.hk

Xinghao Peng
The Hong Kong Polytechnic University
Hong Kong, China
xing-hao.peng@connect.polyu.hk

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Muhui Jiang
The Hong Kong Polytechnic University
Hong Kong, China
muhjiang@polyu.edu.hk

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
hcnzhou@polyu.edu.hk

Yinqian Zhang*
Southern University of Science and Technology
Shen Zhen, China
yinqianz@acm.org

Abstract

Optimistic rollup protocols are widely adopted as the most popular blockchain scaling solutions. As a dominant implementation, Arbitrum has boasted a total locked value exceeding 18 billion USD, highlighting the significance of optimistic rollups in blockchain ecosystem. Despite their popularity, little research has been done on the security of optimistic rollup protocols, and potential vulnerabilities on them remain unknown.

In this work, we unveil three novel double spending attacks on Arbitrum, each enabling an attacker to steal funds from cross-chain applications on Arbitrum. To facilitate these double spending attacks, we introduce an attack to induce manipulable delays in the transaction rollup process and propose a cost optimization solution to reduce further transaction fees associated with the attacks. Our investigations broaden the exploitation of our double spending attacks to another leading optimistic rollup protocol, Optimism, highlighting the generability of our proposed attacks. Through extensive experiments on a local test network, we demonstrated that our attacks lead to severe malicious effects, such as fund losses from double spending. From late 2022 to early 2023, we reported these vulnerabilities to the Arbitrum and Optimism teams. All the issues were acknowledged and resolved, and our research safeguarded billions of dollars at risk, earning us half a million dollars in bug bounty rewards.

*Xiapu Luo and Yinqian Zhang are the corresponding authors.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690256

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Blockchain; Optimistic Rollup; Arbitrum; State Rollback Attack

ACM Reference Format:
Zhiyuan Sun, Zihao Li, Xinghao Peng, Xiapu Luo, Muhui Jiang, Hao Zhou, and Yinqian Zhang. 2024. DoubleUp Roll: Double-spending in Arbitrum by Rolling It Back. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3690256

1 Introduction

Layer-2 (L2) protocols are essential solutions that improve the scalability and transaction throughput of layer-1 (L1) blockchains like Ethereum by executing transactions off the main chain while still benefiting from the underlying blockchain's security. Optimistic rollups have emerged as one of the most popular layer 2 scaling solutions for Ethereum [46], seeing their significant growth and adoption [29]. Their main idea is to execute transactions on the L2 blockchains (rollups) and post transaction data to the L1 blockchain as L1 transactions' payload data [21]. Since the L2 blockchain's execution is based on the submitted transactions to L1 blockchain, anyone can use the transaction data saved on the L1 blockchain to maintain the L2 blockchain and verify the correctness of its state transitions. Leading projects implementing optimistic rollups include Arbitrum[2] and Optimism[4], whose total value locked (TVL) have reached 18B and 7B US dollars[6], respectively.

Since transactions occur off the main blockchain, the security of funds relies heavily on that of rollup protocols. Any vulnerabilities in the rollup protocols can lead to loss of user funds. Unfortunately, there is few work focusing on their security. In this paper, we conduct a thorough analysis of the source code and documentation of Arbitrum and discover a series of vulnerabilities (§6 and §7), which can be exploited by attackers to launch double-spending attacks by strategically triggering state rollback. We also uncovered some similar vulnerabilities in Optimism (§6.4). Although these two popular rollup protocols underwent continuous security audits conducted by well-known security firms [48], none of the vulnerabilities reported in this paper were identified. All of the vulnerabilities we found have been acknowledged and fixed, and our research successfully forestalled potential damages that could have escalated into the billions. In recognition of our pivotal contributions, we were awarded bug bounties by the official security team.

The vulnerabilities we found stem from deficiencies in three aspects, i.e., i) the design of the time bound mechanism used to limit the frequency and extent to which L2 nodes can adjust timestamps[3], ii) the liveness-preservation mechanism used to provide censorship resistance when the sequencer becomes completely unresponsive or even malicious[5], and iii) the transaction (de)compression mechanism used to further reduce transaction costs[1]. More precisely, a transaction sent to Arbitrum will go through first soft finality, when it is included in a L2 block produced by Arbitrum after being executed by Arbitrum Nitro VM, and then hard finality, when it has finality on L1 blockchain (e.g., Ethereum). Since third-party applications (e.g., cross-chain bridges) typically act upon transactions once soft finality has been achieved, if an attacker can force Arbitrum to rollback the state changes caused by the transactions at soft finality, the attacker can successfully launch double-spending attacks. Note that the discovered vulnerabilities can facilitate an attacker accomplishing this malicious aim. For example, an attacker can send a transaction for depositing tokens from Arbitrum to another non-L1 chain via a third-party cross-chain bridge. Once the transaction reaches the soft finality, the bridge will lock the tokens on the Arbitrum side, and mint equivalent tokens on the other chain side, enabling the attacker to use the corresponding tokens within the target chain. Simultaneously, the attacker triggers exploits leveraging the state rollback vulnerabilities, compelling Arbitrum to invalidate the preceding transaction. This, in turn, forces the bridge to release the previously locked tokens to the attacker. Since the equivalent tokens have already been minted on the target chain, the attacker accomplishes double spending.

It is non-trivial to exploit these vulnerabilities because Arbitrum adopts various mechanisms to ensure trustless security. We uncover a new attack vector permitting the injection of manipulable delay to the transactions sent to Arbitrum, making it possible to exploit the aforementioned vulnerabilities (§5). More precisely, our insight is to craft large incompressible data packets and affix them to swathes of L2 transactions in order to deliberately trigger the creation of transaction backlogs. However, this attacking approach may result in a high transaction fee, which is charged by Arbitrum to balance its overhead. To address this challenge, we propose a cost optimization solution to further reduce the attack cost to a reasonable level (§7). Moreover, we find that our attack strategy can also be abused to manipulate posting fees for other users, adversely affecting other Arbitrum users via rising transaction expenses. We further confirm the corresponding double spending vulnerabilities on the other representative Optimistic rollup, i.e., Optimism, highlighting the generality of our proposed attacks.

By conducting experiments in our testbed, we demonstrate the feasibility of launching the double-spending attacks on both Arbitrum and Optimism. Moreover, we conduct extensive experiments to examine the impacts and the expenses associated with each proposed attack. It is worth noting that mitigating these vulnerabilities is non-trivial. Besides disclosing the vulnerabilities to Arbitrum and Optimism, we also suggest remediation solutions for addressing them. Both Arbitrum and Optimism have integrated new components and executed a hard fork of the L2 blockchain to fix them.

In summary, we make the following major contributions.

- We conduct the first in-depth study of state rollback mechanisms on Optimistic rollups, identifying the approaches through which these mechanisms facilitate double spending on these rollups.
- We reveal the delay attacks that can inject manipulable delays into the L2 transactions sent to Arbitrum, making it possible to strategically manipulate and trigger state rollback mechanisms.
- We reveal three types of double spending attacks in Arbitrum, which can cause fund losses of third-party cross-chain contracts on Arbitrum. Specifically, such attacks enable the theft of all funds secured within the cross-chain contracts. We further confirm that our attacks can also threaten other representative Optimistic rollups like Optimism. Per data from [22], a conservative estimate places the value of assets at risk in the billions of dollars.
- We conduct extensive experiments to demonstrate the feasibility of these double-spending attacks and examine their impacts and costs. Our experimental results demonstrate that all attacks can lead to extreme malicious effects, including but not limited to fund losses due to double spending.

2 Background

2.1 Optimistic Rollup Protocol Overview

The optimistic rollup is a kind of L2 protocol designed to boost the throughput of Ethereum transactions. The gist of the optimistic rollup protocol is to publish transaction data on L1 and process transactions on L2, so as to reduce computation on L1. Although various projects have different implementations of the protocol, we abstract the protocol in a unified way to facilitate the description. As illustrated in Figure 1, the L2 ecosystem consists of a L2 node including a sequencer, a batch submitter, and validators, all of which frequently interact with smart contracts deployed on L1.

L2 Node. The L2 node serves as the execution engine for the L2 blockchain, which is built on the Go-Ethereum [23]. One of the key components in the L2 node is the sequencer. A user can create a transaction, sign it with a private key, and send it to the L2 Node's RPC interface. The job of the sequencer is to put the transaction $\mathcal{T}$ received from the RPC interface into an ordered sequence. Once transactions are sequenced, they run sequentially through state transition functions one after another. During this process, the state transition function takes the current state of the chain and a transaction as input to perform the state transition and emits a new L2 block if certain conditions are met. The state transition function is deterministic, meaning the output state depends solely on the current blockchain state and the input transaction.

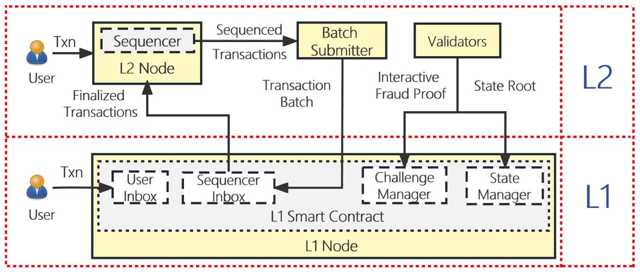

Figure 1: Overview of the optimistic rollup protocol.

The Batch Submitter. A batch $\mathcal{B}$ is formally defined as an ordered sequence of transactions <$\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$...>. The batch submitter is responsible for periodically assembling the sequenced transactions into batches and submitting them to L1. To achieve this, the batch submitter polls the L2 node for the number of transactions that have been sequenced and the sum of the data sizes of these transactions. When the number of transactions or data size exceeds the threshold set in the configuration file, the batch submitter compresses these transactions into a batch and submits the batch to the L1 smart contract named sequencer inbox. Since all transaction data is submitted to L1 in the form of batches, this ensures the data availability of L2. In this way, any user can obtain all transactions of L2 from a L1 archive node and restore the state of the L2 blockchain by applying these transactions sequentially to the genesis state.

The Validator. The validator is responsible for periodically sending the state root of the L2 blockchain to the L1 smart contract named state manager. The state root is defined as the root hash of the current state of the blockchain. Note that the state manager is unable to verify the state root's validity, and a malicious validator can submit an incorrect state root to the state manager. To prevent this, all validators monitor the state roots submitted to the state manager, and if they find an incorrect state root, they will initiate a challenge using the interactive fraud-proof protocol (see appendix A for more details). The fraud-proof protocol ensures that as long as there exists a well-behaved validator, only the correct state root can finally be confirmed by the rollup protocol.

L1 Node. L1 node is an Ethereum node that maintains the contracts used by L2. It is worth mentioning that users can submit transactions that will be executed on L2 to the L1 contract named user inbox. When user inbox receives a transaction, it utilizes the `LOG` opcode to generate an event in the L1 blockchain. The L2 node subscribes to the events generated by the user inbox on L1 and parses them into corresponding transactions to be executed on L2.

2.2 Finality of Transactions and Blocks

Formally, a blockchain $\mathcal{L}$ is defined as an ordered sequence of batches <$\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$...>, $|\mathcal{L}|$ is defined as the number of batches that makes up this blockchain. In the optimistic rollup, the sequencer locally maintains a blockchain $\mathcal{L}_2$, the L1 smart contract maintains a blockchain $\mathcal{L}_1$. The rollup protocol offers two different levels of transaction finality [17]. When the L2 transaction is applied to the state transition function and included in a L2 block, the transaction is considered soft-finalized. When the transaction is soft-finalized, the wallet where the user initiates the transaction will receive the execution result of the transaction and display it to the user. The soft finality of a transaction $\mathcal{T}$ is formally defined as follows:

$$\exists \mathcal{B}, \mathcal{T} \in \mathcal{B} \wedge \mathcal{B} \in \mathcal{L}_2 \wedge \mathcal{B} \notin \mathcal{L}_1$$

Moreover, when L2 transactions are submitted to L1 in the form of batches, and the batch submission transaction is finalized in L1, these L2 transactions are considered hard-finalized. Similarly, if all transactions in a L2 block are hard-finalized, then the L2 block is considered hard-finalized. The hard finality of a transaction $\mathcal{T}$ is formally defined as follows:

$$\exists \mathcal{B}, \mathcal{T} \in \mathcal{B} \wedge \mathcal{B} \in \mathcal{L}_1$$

It is crucial to distinguish the two finality states of a transaction: soft finality and hard finality. When a transaction reaches the hard finality state, its execution result is considered to be final and immutable. However, if a transaction is in the soft finality state, the transaction may still be rolled back under some special circumstances, such as L1 reorganization.

State rollback mechanism. The ideal functionality of state rollback is to ensure the security and correction ability of Optimistic rollup regarding its errors while scaling transaction throughput [19]. When state rollback mechanisms are triggered, transactions that have been soft-finalized on the L2 blockchain will be discarded, and their state modifications will be reverted [19]. For example, when the scenarios, such as L1 blockchain reorganizations [50], differentiate transactions that are hard-finalized on the L1 blockchain and those that are soft-finalized on the L2 blockchain, the L2 node will initiate the rollback of L2 transactions to fix the discrepancies between soft-finalized and hard-finalized transactions.

2.3 Transaction Lifecycle of Arbitrum

Figure 2 depicts the transaction life cycle for L2 transactions on Arbitrum. In summary, the transaction life cycle encompasses two stages. The blue line marks the process of transactions from being accepted to reaching the soft-finalized state, and the pink line shows the process of transactions from the soft-finalized state to reaching the hard-finalized state. Once L2 transactions reach hard-finalized state on L1 blockchain, the L2 node periodically fetches the hard-finalized transactions from the L1 sequencer inbox and compares them with the soft-finalized transactions maintained by the sequencer. If any discrepancies are found, the soft-finalized transactions will be **rolled back** by the L2 node.

– Following the blue line, users submit transactions to RPC service of both the L1 node and the L2 node. The transactions submitted to the L2 node are passed to the sequencer directly. The transactions submitted to the L1 node invoke the user inbox smart contract, and the user inbox will emit an event containing the transaction data. The inbox reader subscribes to the L1 node. Once an event is emitted in the user inbox, the inbox reader will parse the event to the corresponding L2 transaction and deliver it to the sequencer. The sequencer sorts the received transactions and applies state transition functions to these transactions in order. Finally, the state transition function will produce L2 blocks in the soft-finalized state.

– Following the pink line, batch submitter collects sequenced L2 transactions and then compresses and packages them into a batch. The batch is then submitted to the sequencer inbox contract on L1. Once the batch is accepted by the sequencer inbox, the sequencer

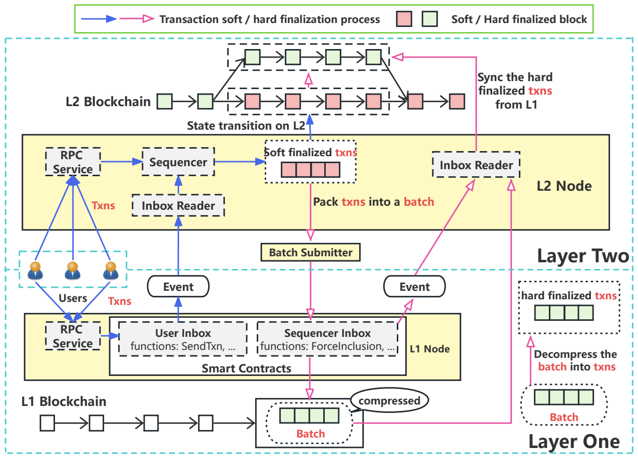

Figure 2: The life cycle of transactions on Arbitrum.

inbox will generate an event representing batch acceptance, and all the transactions in the batch will be in the hard-finalized state. Next, the inbox reader parses the event and downloads the corresponding batch on L1. Afterward, the batch is decompressed into sequenced transactions (in the hard-finalized state), and the L2 node will compare them with those stored in the soft-finalized L2 blocks. Any inconsistency will cause the soft-finalized blocks to be **rolled back**.
Transaction Rollup Process. The ideal functionality of the transaction rollup process is to facilitate the process for L2 transactions to reach the hard-finalized state by submitting L2 transactions to the L1 blockchain in batches. As shown in Figure 2, the transaction rollup process commences with the compression of L2 transactions into batches and concludes upon the reception of these batches by the sequencer inbox on the L1 blockchain.

2.4 Speed Limit Mechanism

In the fraud-proof protocol (see appendix A for more details), when the validator is verifying the state root, the L2 execution engine will run in a virtual machine. One of the security assumptions of the optimistic rollup is that when a validator posts a state root, other validators will check it and respond with a challenge if it is incorrect. This requires that the other validators have enough time and resources to check each state root quickly in order to propose a timely challenge. To prevent the L2 blockchain from processing transactions faster than the validator can simulate them, the execution speed limit mechanism is introduced [15]. This mechanism operates by increasing the L2 gas price when the gas usage exceeds the speed limit (see section 2.5 for more details). Currently, the default L2 gas price stands at 0.1 Gwei for Arbitrum and 0.001 Gwei for Optimism. In addition, the speed limit has been set at 7 million gas per second on Arbitrum and 11 million gas per second on Optimism.

2.5 L2 Transaction Fee Pricing Mechanism

The L2 transaction fee consists of two parts: networking fee and posting fee [13, 35]. The networking fee is akin to the fee paid to miners in L1 and serves as an incentive for blockchain operators, while the posting fee is used to offset the cost for the sequencer to submit transaction data to L1.

2.5.1 *Networking Fee Pricing.*

The networking fee on L2 is calculated based on the L2 gas price and gas usage, and the opcode executed on the EVM of L2 consumes the same gas as that on L1.
Pricing Mechanism on Arbitrum. To prevent the L2 blockchain from processing transactions faster than the validator can simulate them, Arbitrum has implemented an execution speed limit (§2.4). This limit is enforced through the L2 gas pricing algorithm, which tracks the backlog of L2 gas (denoted as B in Formula 1) and increases the L2 gas price exponentially as the backlog grows. Additionally, Arbitrum's gas price adjustment mechanism operates at a one-second granularity, rather than on a per-block basis as in Ethereum. The current speed limit for Arbitrum is 7 million gas per second. According to the white paper of Arbitrum [19], the specific formula is as follow:

$$F(B) = F_0 e^{MAX(0, \beta(B-B_0))} \tag{1}$$

Where F_0 is the minimum L2 gas price (0.1 Gwei) on Arbitrum, and B_0 is a tolerance parameter. When the backlog is less than B_0 ($10 * SpeedLimit$ by default), it will not cause an increase in the L2 gas price. The scaling factor β is deliberately designed such that, during a period of 12 seconds where gas usage is twice the speed limit, it results in the gas price being multiplied by a factor of $\frac{9}{8}$. As a result, β can be calculated numerically as:

$$\beta \approx \frac{1}{102 * SpeedLimit}$$

This matches the base fee growth rate of L1 Ethereum introduced in EIP-1559 [20].
Pricing Mechanism on Optimism. Through analyzing the source code of Optimism's `gas-oracle` component [37], we know that the L2 gas price is adjusted on each epoch (10 seconds) based on the following formula.

$$L2\ gas\ price = current\ gas\ price * factor$$

$$factor = \begin{cases} MIN(1 + \frac{avg\ speed}{speed\ limit}, 1.1) & \text{(avg speed >speed limit)} \\ MAX(1 - \frac{avg\ speed}{speed\ limit}, 0.9) & \text{(avg speed <speed limit)} \end{cases}$$

The L2 gas price is 0.001 Gwei by default [41]. In each epoch, if the average gas usage is greater than the speed limit (11 million gas), the L2 gas price will increase, otherwise, it will decrease. The L2 gas price is limited to a 10% increase or decrease per epoch.

2.5.2 *Posting Fee Pricing.*

Each transaction is submitted to L1 after compression, and the L2 node charges a posting fee to each transaction to offset this overhead. Since L1 gas prices fluctuate, the cost of submitting a batch to L1 also varies dynamically. To accommodate the ever-changing changing L1 gas prices, the posting fee must adjust accordingly to maintain break even. The posting fee is calculated using the following formula.

$$\begin{aligned} posting\ fee &= posting\ units * posting\ unit\ price \\ &\approx (transaction\ size * 16) * posting\ unit\ price \end{aligned}$$

The L2 system dynamically adjusts the posting fee by changing the posting unit price. The posting units are calculated using an algorithm similar to EIP-2028 [7], where each non-zero byte in

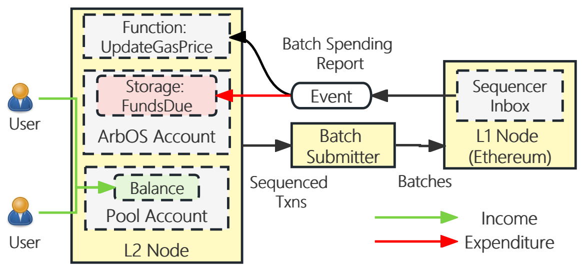

Figure 3: Arbitrum posting fee update process.

the transaction generates 16 units, and a zero byte generates 4 units. Since the transaction data used in the subsequent sections are all random binary data, the number of zero bytes in it is about $\frac{1}{256}$, which is negligible. Therefore, the units corresponding to a transaction can be approximated as 16 times the transaction size.

To adjust the posting fee, Arbitrum uses a special account maintained by the L2 node to record the income and expenditure of the L2 system. Figure 3 shows the process of updating the posting fee in Arbitrum. Specifically, each L2 transaction will be charged an estimated posting fee, which is regarded as an income and automatically transferred to the pool account in the L2 node. Next, the batch submitter publishes a batch to the sequencer inbox and waits for the L1 blockchain to confirm the batch submission transaction. Once the transaction is confirmed, the sequencer inbox will generate an event called batch spending report on the L1 blockchain. After the L2 Node receives the batch spending report, it calculates the cost of submitting the batch and records it as an expenditure in the storage variable named `FundsDue`. Afterward, the `UpdateGasPrice` function is triggered to update the posting fee accordingly. To be specific, `FundsDue` is subtracted from the balance of the pool account as much as possible. Finally, if the deposit in the pool account is reduced to 0, but `FundsDue` is still not settled completely, the posting fee will increase. Otherwise, the posting fee will decrease.

3 Threat model

We assume that a financially rational attacker controls externally owned accounts (EOAs) on both L1 and L2, and these EOAs have sufficient funds. We also assume that all L2 components are not interfered with by any other external factors (such as intrusion detection and system monitoring) during the attack. Since mainstream Optimistic rollups operate in a centralized manner, it is reasonable to assume that the attacker can only interact with L2 by sending transactions like other normal users.

The victims in our threat model are smart contracts on L2 with cross-chain interactions with other chains, which engage in activities outside the L2 blockchain network, such as cross-chain transfers. It is reasonable to assume that these victims rely on the soft-finalized states of L2 transactions to provide their services, as this approach enhances their efficiency and response time. Therefore, attackers can successfully launch double spending attacks in cases where they first initiate transactions to interact with victims (e.g., deposit funds in cross-chain transfers' contracts). Victims then continue their services on other chains under the assumption that the attackers' transactions will eventually be submitted to L1 for hard-finalization. However, attackers trigger state rollback mechanisms on L2 to revert their L2 transactions before they are hard-finalized, thereby causing double spending by obtaining corresponding funds on the other chains without any cost.

4 Mechanisms to trigger rollback

As mentioned in §1, we focus on the double spending attacks caused by state rollback in Arbitrum. To unveil the mechanisms and components in Arbitrum that can trigger state rollback, we opt to investigate the documentation [9] and projects [8] of Arbitrum related to the life cycle of transactions. This choice is made because state rollback can occur at various stages of the process of handling transactions until the transactions are hard finalized. Finally, we have identified three specific mechanisms that can trigger state rollback, which we will discuss in detail in the following.

4.1 Time Bound Mechanism.

During the transaction synchronization process, the L2 node can adjust the timestamp of the transactions in soft finality to account for those delays and prevent any potential reorganizations of the chain [3]. The proposed time-bound mechanism in Arbitrum aims to restrict the frequency and extent to which L2 nodes can adjust timestamps [3]. This limitation is crucial to safeguard decentralized applications and services that rely on time constraints, as manipulation of transaction timestamps can render them vulnerable to attacks [18]. Currently, Arbitrum sets the time boundary for adjusting transaction timestamps to 24 hours. Specifically, on Arbitrum, the timestamp of the received batch is checked in the L1 sequencer inbox. To be specific, the timestamps of all L2 transactions corresponding to this batch $\mathcal{B}$ must satisfy the following constraint where the maximum variation $\mathcal{V}$ is equal to the submitter-only window.

$$\forall \mathcal{T} \in \mathcal{B}, \mathcal{T}_{L2.Timestamp} >= \mathcal{T}_{L1.Timestamp} - \mathcal{V}$$

If the transaction's L2 timestamp is less than the L1 timestamp minus the maximum variation, then the L2 timestamp will be corrected to the boundary value (i.e. $\mathcal{T}_{L1.Timestamp} - \mathcal{V}$) by the L2 node in the transaction synchronization mechanism [16].

How to roll the state back: If the delay time of transactions in submitted batches surpasses $\mathcal{V}$, the L2 node will update its timestamps. In such cases, soft-finalized transactions maintained by L2 node will be rolled back due to inconsistency between the original transactions in submitted batches and those whose timestamps have been modified.

4.2 Liveness-Preservation Mechanism

The liveness-preservation mechanism in Arbitrum is used to provide censorship resistance when L2 components become completely unresponsive or even malicious [5]. Specifically, in the liveness-preservation mechanism, even if the L2 components, such as the sequencer, stop working, users can still submit cross-chain transactions from L1 directly and force them to be included to keep the liveness of L2 blockchain.

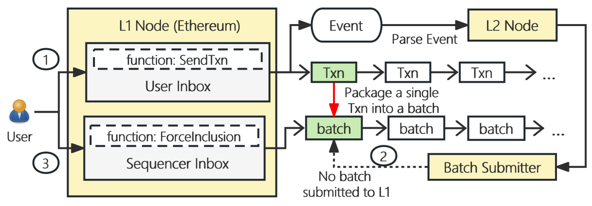

Figure 4: The liveness-preservation mechanism.

Figure 4 demonstrates the workflow of the keeping liveness mechanism. In the first step, the user sends a cross-chain transaction to the L1 user inbox, and the user inbox emits an event after accepting the transaction. Next, the L2 node parses the L1 event and mints the corresponding L2 transaction. The minted L2 transaction will be executed and packed into a L2 block. In the second step, the batch submitter packs L2 transactions (including the minted L2 transaction) into a batch and submits the batch to L1. The above steps assume that all L2 components are working properly, and the minted L2 transactions will be automatically packaged into a batch and uploaded to L1 without any further user interaction. However, if the L2 network is down, the user must call a special function in the sequencer inbox to force the inclusion of the cross-chain transaction. As shown in the third step, the user calls the function defined in the sequencer inbox to package the L2 transaction into a batch without involving any L2 component. Since the L2 transaction is saved into a batch, it is considered hard-finalized. The force inclusion of cross-chain transactions provides a guarantee of the liveness of the L2 blockchain. Even if the centralized L2 components, such as the sequencer, stop working, users can still submit cross-chain transactions from L1 and force them to be included to keep the L2 blockchain running.

Note that the force inclusion feature is only allowed to be used after a certain delay (denoted by $\mathcal{V}_L$) once the cross-chain transaction has been accepted by the user inbox. During the delay time, only the batch submitter is allowed to submit batches containing L2 transactions to L1. The purpose of this delay is to avoid conflicts between the force inclusion mechanism and the batch submitter. Currently, Arbitrum sets the delay time $\mathcal{V}_L$ to 24 hours.
How to roll the state back: If the delay time of a cross-chain transaction originating from L1 surpasses $\mathcal{V}_L$, and the transaction is force-included, soft-finalized transactions maintained by L2 node will be rolled back due to inconsistency between the force-included transaction and the transaction submitted by batch submitter.

4.3 Transaction (de)compression mechanism

During the transaction synchronization between L1 and L2, the batches submitted from L2 are compressed to further reduce transaction costs on L1, as discussed in §2. Following inclusion in L1, the compressed batches are retrieved by the L2 node, which then decompresses them into sequenced transactions within the hard-finalized state. Subsequently, these decompressed transactions are used to determine transactions within L2, achieved through comparison with those stored in the soft-finalized L2 blocks. For example, the L2 node will discard soft-finalized transactions that do not align with the decompressed transactions of the hard-finalized state.

Note that, during the decompression process, to prevent an overload of data leading to a denial-of-service situation for the L2 node, Arbitrum deems a compressed batch in L1 as invalid if its decompressed data exceeds the upper limit size.
How to roll the state back: When an invalid batch occurs, the soft-finalized transactions maintained by the L2 node will be rolled back due to inconsistency with the transaction data in the batch.

5 Manipulable Delay attack

We further uncover a novel delay attack permitting the injection of manipulable delay of arbitrary duration to the transactions sent to Arbitrum, making it possible to exploit the aforementioned mechanisms to trigger state rollback. Specifically, the successful launch of double spending attacks crucially relies on the intentional initiation of state rollback for our target L2 transactions (e.g., deposit transactions), achieved by triggering the three corresponding mechanisms delineated in §4. Note that, the activation of these mechanisms is contingent upon the satisfaction of non-trivial and unusual conditions. For example, the time-bound mechanism and the liveness-preservation mechanism mandate the existence of a transaction with delay time surpassing $\mathcal{V}$ and $\mathcal{V}_L$, respectively. Therefore, the attacker can exploit our delay attack to manipulate the delay duration of the targeted transactions, ultimately successfully triggering state rollback and launching double spending attacks.

Our strategy to launch the delay attack is to delay the transaction finality time. If the time for the transaction to reach hard finality is delayed, the transaction remains in the soft finality state, making it susceptible to rollback during the state rollback process. To delay the transaction finality time, our insight is to create the batch backlog. Figure 5 demonstrates the workflow of a successful delay attack leveraging the batch backlog. First, the attacker constructs large-sized incompressible data (e.g., random binary data [45]), appends the data to a large number of L2 transactions, and then sends these transactions to the L2 node. These L2 transactions are accepted by the L2 blockchain instantly because Arbitrum adopts a one-transaction-per-block or similar strategy. Next, the batch submitter tries to compress these transactions into batches. However, since these transactions contain incompressible data, the size of these transactions remains basically unchanged before and after the compression. In addition, since the size of these transactions is very large (more than half of the batch size limit), each transaction is packed into a separate batch. Finally, the batch submitter will publish these batches to L1 one by one. Since the block interval of L1 is about 12 seconds and the batch submitter is designed to wait for L1 confirmation, it will take a lot of time to publish these batches to L1 when the number is large. In a word, an adversary can send a large number of incompressible L2 transactions to the L2 node, and the batch submitter cannot submit the batches to L1 in a short period of time, thus creating a batch backlog. The number of backlogged batches can be formally defined as $|\mathcal{L}_2| - |\mathcal{L}_1|$.

Due to the batch backlog, any L2 transactions submitted by users cannot be published to L1 in a timely manner, including our target transactions to launch double spending. This also means that these user transactions cannot reach the hard finality state for a long time after they reach the soft finality state.

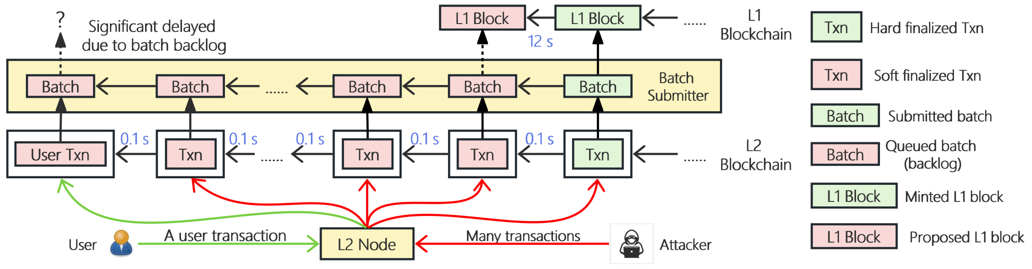

Figure 5: The workflow of the delay attack leveraging the batch backlog.

6 Three Double Spending attacks

In this section, we introduce three double-spending attacks. Each of them exploits the delay attack (§5) to trigger L2 rollbacks through distinct mechanisms mentioned in §4. Specifically, the three double spending attacks, i.e., overtime attack (§6.1), QueueCut attack (§6.2), and zip-bomb attack (§6.3) target the three mechanisms for triggering state rollback, i.e., time bound mechanism (§4.1), liveness preservation mechanism (§4.2), and transaction (de)compression mechanism (§4.3), respectively. The fundamental strategy of the three attacks involves preparing a withdrawal transaction that achieves initial soft finalization on L2 but is later reverted due to state rollback. This reversal permits the transaction's re-execution, thereby enabling double spending. The subsequent subsections elaborate on the procedures of these attacks.

6.1 Overtime Attack

The overtime attack employs the delay attack (§5) to induce a delay in the transaction rollup process from L2 to L1 (§2.3). This orchestrated delay ultimately activates a time-bound correction, leading to rollbacks on L2. As mentioned in §4.1 about the time-bound mechanism, transactions failing to be submitted within the submission window, denoted as $\mathcal{V}$, will be reverted. The overtime attack, therefore, involves deliberately causing a delay of $\mathcal{V} + \Delta$, where Δ represents the duration within which we initiate a cross-chain deposit transaction destined for reversal. Consequently, the batch submitter is unable to submit this transaction within the timeframe of $\mathcal{V}$, thereby activating the time-bound mechanism to roll back L2. As a result, transactions that were soft-finalized on L2 are reverted, enabling the attacker to execute a withdrawal on the other end of cross-chain bridge without actually depositing assets.

Figure 6 presents the attack steps of the overtime attack. We denote the cross-chain deposit transaction as $Tx_{deposit}$, and the delay-inducing transactions as Tx_{delay}. This attack unfolds over four steps. ① Initially, the attacker dispatches a sufficient number of Tx_{delay} to provoke a delay totaling $\mathcal{V} + \Delta$ (noting that $\mathcal{V}$ is the window for submission and Δ is the timeframe allocated for submitting $Tx_{deposit}$ to L2). ② Subsequently, in the second step, attacker submits $Tx_{deposit}$ to L2 within the Δ period. ③ During the third step, the L2 sequencer accepts $Tx_{deposit}$, yet the batch submitter's attempt to relay the batch containing $Tx_{deposit}$ to the L1 sequencer inbox is thwarted, failing to meet the $\mathcal{V}$ deadline due to the induced delay. Concurrently, the attacker initiates a withdrawal on the target chain. ④ The fourth step sees the time-bound mechanism spring into action, recognizing that $\mathcal{T}_{L1.Timestamp}$ surpasses $\mathcal{T}_{L2.Timestamp} + \mathcal{V}$. It adjusts $\mathcal{T}_{L2.Timestamp}$ to $\mathcal{T}_{L1.Timestamp} - \mathcal{V}$, triggering a rollback of L2's state. Here, $Tx_{deposit}$ is reverted. However, in step ③, the assets have already been withdrawn from the target chain. Consequently, the assets are minted on the target chain at no cost, and the deposited assets are returned, thus leading to a double spend.

6.2 QueueCut Attack

The QueueCut attack, akin to the overtime attack, initially induces a delay in batch submission from L2 to L1, resulting in a L2 transaction queue where each transaction is softly finalized but awaits hard finalization. We then exploit the liveness-preservation mechanism to strategically insert a transaction at the front of the queue. Consequently, the L2 queue must realign with L1, leading to the rollback of these softly finalized transactions in L2 (§2.3). To capitalize on this process, we orchestrate a cross-chain deposit transaction within this queue, similar to the overtime attack. This attack successfully minted assets on the target chain, but the deposited assets were returned on Arbitrum, achieving a double spend.

Figure 7 illustrates the attack workflow of the QueueCut attack. ① Initially, the attacker introduces a delay of $\mathcal{V} + \Delta$ in the L2-to-L1 submission process by dispatching a sufficient number of transactions containing incompressible data (Tx_{delay}). ② Subsequently, in the second phase, the attacker launches a cross-chain deposit transaction $Tx_{deposit}$ that achieves soft finalization on L2 instantly. This allows the attacker to commence a withdrawal transaction on the target chain. ③ In the third phase, the attacker transmits an L1-L2 transaction (Tx_{L1-L2}) to the L1 contract UserInbox. Note that both $Tx_{deposit}$ and Tx_{L1-L2} will not reach hard finalization within $\mathcal{V}$ due to the previously induced delay. Nevertheless, these transactions create a queue of soft-finalized transactions awaiting sequential hard finalization. ④ Proceeding to the fourth step, after the $\mathcal{V}$ period, the attacker can call the ForceInclusion function in the L1 contract sequencer inbox to hard-finalize the L1-L2 transaction. The forced inclusion of the Tx_{L1-L2} lets it cut the queue of transactions waiting to be hard-finalized. Consequently, the initial soft finalized queue becomes misaligned with the hard finalized version, leading to its rollback. The remaining transactions, including $Tx_{deposit}$, are reverted and deleted. The deposited assets are returned to the attacker. However, the attacker has already withdrawn them from the target chain. This results in double spending.

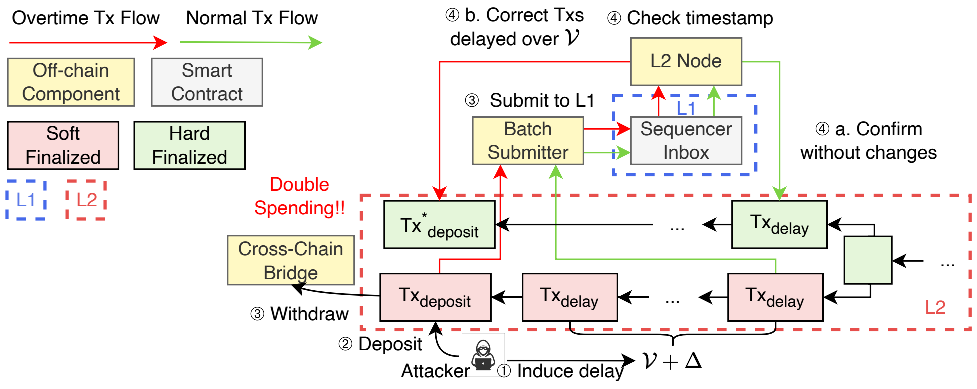

Figure 6: The workflow of overtime attack.

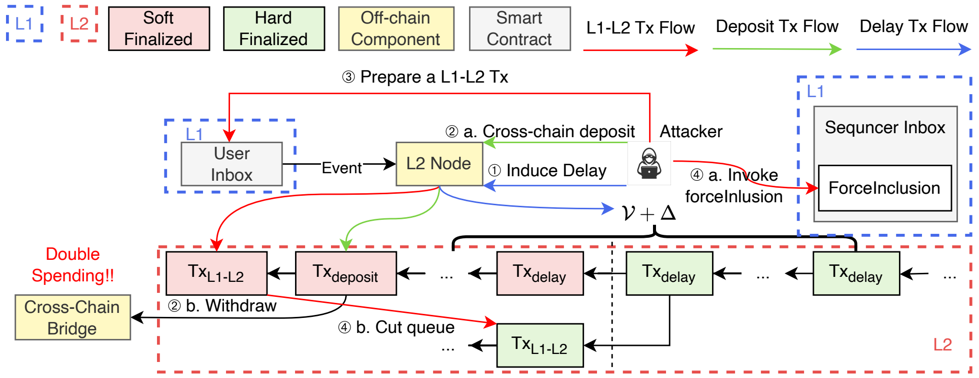

Figure 7: The workflow of QueueCut attack.

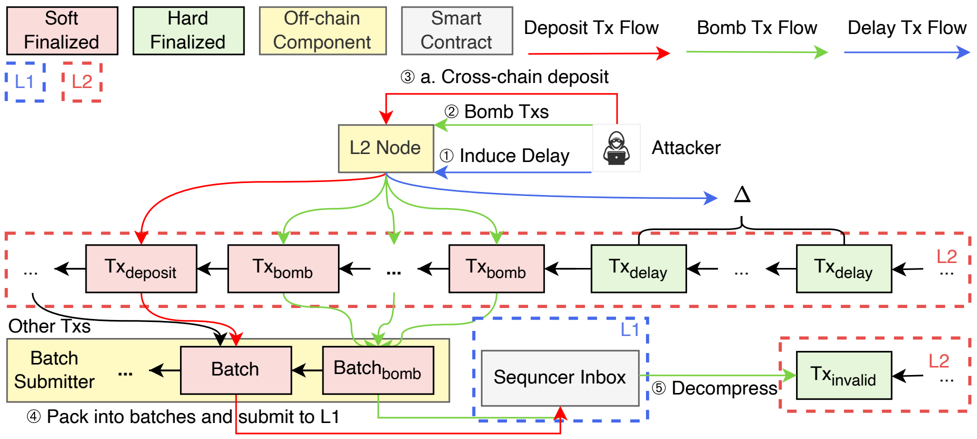

Figure 8: Launch a double spending attack via zip-bomb.

6.3 Zip-Bomb Attack

The zip-bomb attack exploits a flaw in the transaction (de)compression mechanism, causing state rollback and enabling double spending. As mentioned in §4.3, transactions on L2 are compressed and grouped into batches before being submitted to L1. When an L2 node retrieves these batches from L1, it must decompress them. This flaw arises because there is a size-bound check during decompression but not during compression. Consequently, it is possible to create a batch containing enough transactions to exceed the size bound during decompression. This flaw can be exploited to revert a cross-chain deposit transaction, achieving double spending.

The exploit steps of this zip-bomb attack are outlined in Figure 8. ① The attacker first induces a delay, Δ, by sending incompressible transactions (Tx_{delay}). ② In the second stage, the attacker sends a series of transactions (Tx_{bomb}) with zero-padded data. ③ In the third stage, the attacker sends a cross-chain deposit transaction. ④ In the fourth stage, the batch submitter compresses the transactions and packs them into batches before submitting them to L1. Notably, Tx_{bomb} transactions have a very high compression rate due to the zero-padded data, allowing a large number of Tx_{bomb} to be packed into a single batch ($Batch_{bomb}$). Additionally, the induced delay Δ means the batch is not immediately submitted to L1. Before the batch reaches L1, the cross-chain bridge processes the deposit and mints tokens on the target chain. After Δ, $Batch_{bomb}$ reaches L1. ⑤ In the fifth stage, the L2 node retrieves $Batch_{bomb}$ from L1 and decompresses it. Since the size of the decompressed data exceeds the limit, the L2 node deems it to contain invalid transactions ($Tx_{invalid}$). This misalignment with the soft-finalized transactions on L2 leads to their reversal, including $Tx_{deposit}$. Therefore, the assets deposited are returned to the attacker, while the tokens have already been minted on the target chain, leading to double spending.

6.4 Double Spending on Optimism

After investigating the design, documentation, and implementation of another representative Optimistic rollup, i.e., Optimism [30], we find that partial of our proposed attacks (i.e., the manipulable delay attack in §5 and the QueueCut attack in §6.2) can also be launched on it. This is because Optimism employs the same mechanisms (i.e., transaction rollup mechanism and liveness-preservation mechanism) as Arbitrum, which are the corresponding attack targets of the delay attack and QueueCut attack. Please note that by leveraging the above two attacks, an adversary can successfully conduct the double spending on Optimism, highlighting the generability of our proposed attacks.

7 Attack Cost Optimization

The cost for launching the double spending attacks mainly consists of the transaction fees used for launching the delay attack (§5), because i) the transactions we construct for attacks are mostly involved in delay attack, and ii) the funds we deposit during the attacks will withdraw at the end of double spending. However, due to the transaction fee pricing mechanism (§2.5), the per-transaction fee may continue to increase during the delay attack, rendering it economically infeasible for a financially rational attacker. To address this challenge, we propose a cost optimization solution to further reduce the attack cost to a reasonable level (e.g., a constant cost). The main idea of our cost optimization solution is to i) manipulate the posting unit price to reduce the posting fee of our-initiated transactions on L2, and ii) submit transactions at a speed lower than the speed limit mechanism (§2.4) to avoid an increase in the networking fee. In the following, we will first introduce the posting unit pricing algorithm on Arbitrum (§7.1) and then explain how to manipulate the posting unit price (§7.2) and control the transaction submitting frequency (§7.3).

7.1 Arbitrum Posting Unit Pricing Algorithm

We first formulate the price update algorithm on Arbitrum for the posting unit price. According to the white paper of Arbitrum [19], the posting unit price is updated based on the below formulas.

$$P = MAX(0, P_{prev} + \Delta P) \tag{2}$$

Formula 2 is utilized to determine the current posting unit price. The variables used in this equation are defined as follows: P represents the current posting unit price; P_{prev} denotes the posting unit price during the previous update; and ΔP indicates the change in price, which could be either positive or negative.

$$\Delta P = (D' - D)\frac{U}{\alpha + U} \quad (U > 0) \tag{3}$$

Formula 3 is used to calculate the price change. The variables utilized in this formula are defined as follows: D' is the derivative that must hold on average in order for the surplus to reach zero after processing E more data units. D represents the derivative of the surplus; U denotes the posting units assigned to this update; and α is a smoothing parameter, which is defined as 1.6×10^7.

$$D = \frac{\Delta S}{U} = \frac{S - S_{prev}}{U} \quad (U > 0) \tag{4}$$

$$D' = -\frac{S}{E} \tag{5}$$

Formulas 4 and 5 are used to calculate the two derivatives. The variables used in these formulas are defined as follows: S represents the current surplus, which could be either positive or negative; S_{prev} denotes the surplus in the previous update; and E indicates the equilibration constant, which is defined as 1.6×10^8.

$$U = U_{total} \cdot \frac{T_{upd} - T_{prev}}{T_{current} - T_{prev}} \tag{6}$$

Formula 6 is used to compute data units assigned to the current update. The variables used in this equation are defined as follows: U_{total} is the number of data units that remain unassigned; $T_{current}$ represents the current time; T_{upd} denotes the time at which the current batch was submitted; and T_{prev} indicates the time at which the previous batch was submitted. The intuition behind this formula is that the L2 node knows the number of units collected between T_{prev} and $T_{current}$, and it wants to estimate how many units were collected between T_{prev} and T_{upd}.

Assuming that the batch submission interval and the L1 block interval are fixed, then $\frac{T_{upd}-T_{prev}}{T_{current}-T_{prev}}$ can be regarded as $\frac{1}{K}$. Therefore, formula 6 can be rewritten as follows, where U_n represents U at n-th update and U_{total}^{n-1} represents U_{total} at (n-1)-th update.

$$U = \frac{U_{total}}{K} \implies U_n = \frac{U_{total}^{n-1} + \Delta U}{K}$$

Assuming that the L2 blockchain reaches a stable state in the long run, then ΔU remains constant across price updates, leading to U_{total}^{n} and U_n being approximately equal to U_{total}^{n-1} and ΔU, respectively. To simplify calculations, we will approximate U as ΔU. In other words, we will consider U to be the units that correspond to the transactions submitted within a price update cycle.

7.2 Manipulate The Posting Fee

Our insights to manipulate the posting fee are based on the following two observations. First, the pool account is a standard externally owned account (EOA), and the income of the L2 system is stored in its balance. This gives an opportunity for an adversary to manipulate the income of the L2 system by directly transferring funds to the pool account. Second, an adversary can leverage the delay attack to make the interaction between L2 and L1 change from synchronous to asynchronous, thus disrupting the posting fee pricing process. Hence, our solution consists of two steps for optimizing the posting fee during the delay attack.

– *Step 1.* The attacker directly transfers a substantial amount of funds to the pool account, ensuring a high surplus in the pool account. Once a new batch spending report is received in the L2 node, the `UpdateGasPrice` function is triggered. Since the pool account has a high surplus, the posting unit price can be updated to nearly zero according to the price update algorithm (§7.1).

– *Step 2.* The attacker sends a large number of transactions containing incompressible data to launch the delay attack. Since the posting unit price has been down to nearly zero, the posting fee of these transactions are significantly reduced.

Additional attack impacts. After the attacker stops submitting transactions, the posting fee will increase from nearly zero to a significant number, consequently raising the cost for other users to send transactions on L2. Specifically, during the attack, the attacker creates a large backlog of batches that cannot be submitted to L1 in a timely manner, causing a significant delay in the price update process. In such cases, after the attacker stops sending transactions containing incompressible data to the L2 node, the backlog of batches will be gradually submitted to L1. A batch spending report event is generated after each batch is accepted by the sequencer inbox in L1. After the L2 node receives this event, it will obtain the batch submission cost and record it as an expenditure of the L2 system in the `FundsDue` storage variable and then call the `UpdateGasPrice` function to adjust the posting unit price. During this process, the value recorded in `FundsDue` will continue to increase, which means that the L2 system owes more and more money, so each call to `UpdateGasPrice` function will cause the posting unit price to increase. As long as the backlog of batches is large enough, the posting unit price can rise to any high point when the backlog of batches is released. At this time, the cost for a user to send a transaction on L2 will be very high or even far exceed the transaction cost on L1. In §8.4, we will show experimentally that the posting unit price can escalate to over 14,000 Gwei, representing a 700-fold increase from the gas price in L1.

7.3 Transaction Submitting Frequency

To avoid the increased cost due to the rapid growth of the networking fee (cf. speed limit mechanism in §2.4 and networking fee pricing in §2.5.1), an attacker can only launch the attack at a limited rate. The upper limit of the batch size in Arbitrum is 100,000 bytes [14]. Hence, an attacker can send malicious transactions containing more than 50,000 bytes of incompressible data. Since the size of any two adjacent transactions exceeds the upper limit of the batch size, each transaction will be packaged separately as a batch. According to EIP-2028 [7], every non-zero byte in a transaction costs 16 gas. Since the transaction sent by the attacker contains random binary data, the number of zero bytes in it is about $\frac{1}{256}$, which is negligible. Given that the speed limit on Arbitrum is set at 7 million gas per second, on the premise that the gas price does not increase, the frequency limit for the attacker to submit batches is 8.75 per second, which is calculated as follows.

$$\begin{aligned} batch\ submit\ frequency &= \frac{speed\ limit}{batch\ size * gas\ cost\ per\ byte} \\ &= \frac{7,000,000}{50,000 * 16} = 8.75 \end{aligned} \tag{7}$$

As the speed limit on Optimism is 11 million gas per second, by applying the equation 7, it can be seen that under the premise of ensuring that the L2 gas price does not increase, the highest frequency of submitting batches is 15.28 per second on Optimism.

8 Evaluation

Feasibility of the proposed attacks.

- For double spending attacks on Arbitrum, we reproduced the three kinds of double-spending attacks by triggering corresponding mechanisms for state rollback in our local test blockchain.
- For the manipulable delay attack on Arbitrum, we reproduced the vulnerabilities on our local blockchain under the cost optimization solution.
- For double spending attacks on Optimism, we reproduced the QueueCut attack to conduct double spending by triggering the liveness-preservation mechanism.
- For the manipulable delay attack on Optimism, we reproduced the vulnerabilities on our local blockchain.

We further conduct experiments to answer the following four research questions for evaluating the costs and impacts of our proposed attacks. **RQ1:** What are the costs and impacts of the delay attack on Arbitrum? **RQ2:** What are the costs and impacts of the delay attack on Optimism? **RQ3:** How does the delay attack threaten L2 in practice? **RQ4:** What are the costs and impacts brought by the cost optimization solution?

Experimental Setup. To record runtime data such as batch confirmation time and transaction cost, we perform code instrumentation on the L2 node and batch submitter. We also patched Arbitrum to allow it to connect to a private L1 blockchain. Our experiments are performed on two virtual machines running Ubuntu 22.04, each with 4 vCPU and 10 GB RAM. To simulate the real-world scenario, we leverage Hardhat [24] to run a local L1 blockchain that maintains a fixed block generation interval of 12 seconds and a fixed L1 gas price of 20 Gwei. For experiments about Arbitrum, an Arbitrum node is installed on the virtual machine. For experiments about Optimism, we use the officially provided docker file [40] to run a local L2 blockchain.

8.1 Delay attack impacts and costs on Arbitrum

Delay attack impacts. As calculated in §7.3, the maximum batch submitting frequency on Arbitrum is 8.75 per second, and the batch submission interval on Arbitrum is about 25 seconds. Hence, launching the attack for every second can result in an average delay of $8.75 * 25 - 1 = 217.75$ seconds.

Cost on the Transfer Amount. As mentioned in §7.2, the cost of delay attack consists of two parts, i.e., the funds transferred to the pool account on Arbitrum, and the cost to create a batch backlog. Assume that the current system is in a balanced state, that is, the balance in the pool account and the value in `FundsDue` are both 0. As a result, formula 3 can be rewritten as follows.

$$\begin{aligned}\Delta P &= (D' - D)\frac{U}{\alpha + U} && \text{(s.t. } U > 0\text{)}\\ &= -(\frac{S}{E} + \frac{S - S_{prev}}{U})\frac{U}{\alpha + U}\\ &= -(\frac{S}{E} + \frac{S}{U})\frac{U}{\alpha + U} && \text{(Assume } S_{prev}\text{=0)}\\ &= -\frac{S(E+U)}{E(\alpha + U)}\end{aligned} \tag{8}$$

Suppose an attacker wants to manipulate the posting unit price to 0 in one price update. In this case, ΔP is a negative number. When FundsDue is 0, the surplus equals the deposit in the pool account, which corresponds to the amount transferred by the attacker. At this time, S can be expressed as $S = T - \Delta P \cdot U$ where $P_{current} + \Delta P = 0$. This means that the current posting unit price will become 0 after a price update. Under this condition, formula 8 can be rewritten as follows.

$$\Delta P = -\frac{S(E+U)}{E(\alpha + U)} = -\frac{(T - \Delta P \cdot U)(E+U)}{E(\alpha + U)} \tag{9}$$

From this formula, we can calculate how much surplus is needed to manipulate the posting unit price to 0. Assuming that the L1 gas price is 20 Gwei, the result is calculated in equations 10.

$$S_1 = -\Delta P \cdot U = -\Delta P \cdot \sqrt{E\alpha} \approx 1.01\ ETH \quad (T = 0) \tag{10}$$

Equation 10 can be derived from formula 9 by setting $T = 0$. To summarize, to manipulate the posting unit price to zero, the attacker should send 0.32 ETH to the pool account.

Cost on Creating Batch Backlog. We proceed with an analysis of the cost of continuously generating the batch backlog. At the onset of the attack, a surplus was present in the L2 system since the attacker had transferred some funds to the pool account to manipulate the posting unit price. Due to the existence of surplus, D' will be a negative number, D is relatively small and can be ignored, so ΔP is a negative number, and P is always 0. This process will persist until S is close to 0. Afterward, the posting unit price will rise slightly and then fluctuate at an equilibrium price. At the equilibrium price, the posting fees for all transactions in a price update cycle (about 25 seconds) are equal to the cost of submitting a batch to L1. In other words, S and ΔS are basically kept around 0 in each price update when the current posting unit price is equal to the equilibrium price. According to equation 12, the cost of submitting a batch that contains 50,000 bytes on Arbitrum can be computed as 0.016 ETH, which aligns with the surplus reduction value during each price update cycle. Assuming the attacker submits transactions at a rate of two per second, the equilibrium price can be calculated as follows.

$$\begin{aligned}P_{equilibrium} &= \frac{batch\ submission\ cost}{txn\ size * txn\ quantity * gas\ cost\ per\ byte}\\ &= \frac{0.016\ ETH}{50,000 * (2 * 25) * 16}\\ &= 0.4\ Gwei\end{aligned} \tag{11}$$

There are also networking fees for sending transactions on L2. Hence, the cost of submitting a L2 transaction with incompressible data can be calculated as follows.

$$\begin{aligned}cost &= txn\ size * gas\ cost\ per\ byte * gas\ price\\ &= 50,000 * 16 * (0.4\ Gwei + 0.1\ Gwei)\\ &= 0.0004\ ETH\end{aligned}$$

8.2 Delay attack impacts and costs on Optimism

Delay attack impacts. As calculated in §7.3, the maximum batch submitting frequency on Optimism is 15.28 second. Therefore, every second of attack launched can cause a delay of $15.28*12-1 = 182.36$ seconds on average.

Delay attack costs. Optimism adopts a relatively simple mechanism to calculate the posting fee [42], as opposed to Arbitrum's complex break-even maintenance system. Specifically, Optimism dynamically adjusts the posting unit price by tracking the current gas price on L1 Ethereum. Therefore, we can approximate the posting unit price as the gas price on L1 Ethereum to facilitate the calculation. Since the gas price of L2 is much lower than that of L1, the networking fee is almost negligible compared to the posting fee. As a result, the cost of submitting an incompressible transaction can be approximated using the following formula.

$$\begin{aligned}cost &= batch\ size * gas\ cost\ per\ byte * posting\ unit\ price * 1.5\\ &= \frac{max\ batch\ size}{2} * gas\ cost\ per\ byte * L1\ gas\ price * 1.5\\ &= 45,000 * 16 * 20\ Gwei * 1.5 = 0.0216\ ETH\end{aligned} \tag{12}$$

It is known that the maximum batch size of Optimism is 90,000 bytes [36], so only transactions with a size exceeding 45,000 bytes will be packaged into a single batch. In addition, the L1 gas price is assumed to be 20 Gwei, and Optimism's L2 node sets the posting unit price to 1.5 times the actual L1 gas price. Therefore, the cost of submitting a L2 transaction with incompressible data is 0.0216 ETH. Since each such transaction will be packaged into a separate batch, Optimism needs to wait for a batch to be confirmed by L1 before submitting the next one (it takes about 12 seconds). The cost to cause a one-second delay is 0.0216 ETH / 12 = 0.0018 ETH.

8.3 Effectiveness of delay attack

In this subsection, we demonstrate the effectiveness of delay attacks on Arbitrum and Optimism in practice. Figure 9 illustrates the effectiveness of the manipulable delay attack. It can be seen that the average batch submission interval is approximately 25 seconds and 12 seconds on Arbitrum and Optimism, respectively. This implies that for each backlogged batch, a delay of 25 seconds on Arbitrum and 12 seconds on Optimism is incurred. Moreover, the output of the instrumented code shows that the cost of the delay attack on Optimism is constantly at 0.0217 ETH per batch (each batch will cause a delay of 12 seconds), which is consistent with the theoretical result calculated in formula 12.

8.4 Cost and impacts for cost optimization

Since the cost for our cost optimization solution has been explored in §8.1, we will reuse the corresponding results. Besides, as mentioned in §7, the cost optimization will bring additional attack impacts, which we will detail in the following.

Assuming that U is the same in each price update cycle and $U > 0$, the final posting unit price can be calculated by the recursive formula 13. In that formula, n represents the number of price update cycles, and we assume that $P_0 = 0$, $S_0 = 0$, and $C = batch\ submission\ cost = 0.016\ ETH$. The theoretical results are shown later in this subsection.

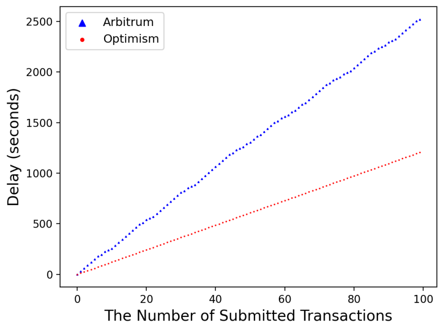

Figure 9: The effectiveness of the delay attack on Arbitrum and Optimism.

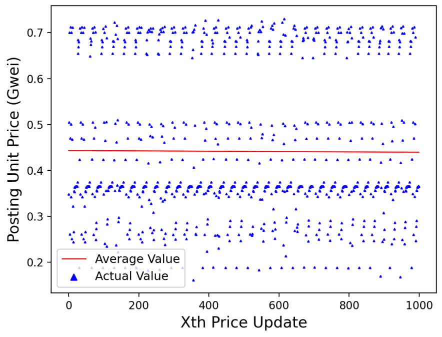

Figure 10: The cost caused by the cost optimization solution on Arbitrum.

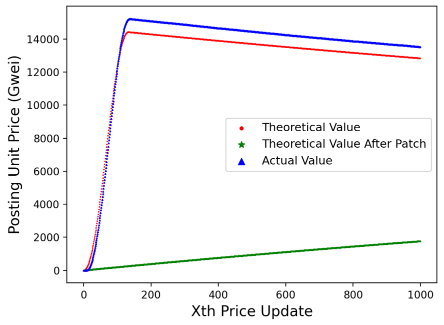

Figure 11: Additional attack effectiveness.

$$\begin{aligned}P_n &= P_{n-1} + \Delta P_{n-1}\\ &= P_{n-1} - (\frac{S_{n-1}}{E} + \frac{\Delta S_{n-1}}{U})\frac{U}{\alpha+U}\\ &= P_{n-1} - \frac{U}{\alpha+U}(\frac{1}{E}\sum_{i=1}^{n-1}\Delta S_{i-1} + \frac{\Delta S_{n-1}}{U})\\ &= P_{n-1} - \frac{U}{\alpha+U}[\frac{1}{E}\sum_{i=1}^{n-1}(P_{i-1}\cdot U - C) + \frac{(P_{n-1}\cdot U - C)}{U}]\end{aligned} \tag{13}$$

Interestingly, we discovered an inconsistency between the implementation of Arbitrum for updating posting unit prices and its white paper. Due to the implementation error, when storing the surplus in the database, Arbitrum performs an implicit type conversion that takes the absolute value of the surplus. Consequently, when the surplus becomes negative, the formula used for calculating the price incorrectly changes to the following:

$$\begin{aligned}P_n &= P_{n-1} + \Delta P_{n-1}\\ &= P_{n-1} - (\frac{S_{n-1}}{E} + \frac{S_n - |S_{n-1}|}{U})\frac{U}{\alpha+U}\\ &= P_{n-1} - \frac{U}{\alpha+U}(\frac{S_{n-1}}{E} + \frac{S_n + S_{n-1}}{U})\\ &= P_{n-1} - \frac{U}{\alpha+U}(\frac{S_{n-1}}{E} + \frac{2*S_{n-1} + \Delta S_{n-1}}{U})\\ &= P_{n-1} - \frac{U}{\alpha+U}(\frac{2E+U}{EU}\cdot S_{n-1} + \frac{\Delta S_{n-1}}{U})\\ &= P_{n-1} - \frac{U}{\alpha+U}[\frac{2E+U}{EU}\sum_{i=1}^{n-1}(P_{i-1}\cdot U - C) + \frac{(P_{n-1}\cdot U - C)}{U}]\end{aligned} \tag{14}$$

It can be seen that formulas 13 and 14 have different coefficients $\frac{1}{E}$ and $\frac{2E+U}{EU}$ for the S_{n-1} term. Since $\frac{2E+U}{EU}$ is much larger than $\frac{1}{E}$, this implementation error will greatly amplify the effect of the attack. To assess the relative magnitudes of the two coefficients in detail, we can compute their ratio as follows:

$$\frac{\text{coefficient1}}{\text{coefficient2}} = \frac{\frac{2E+U}{EU}}{\frac{1}{E}} = \frac{2E+U}{U} = \frac{2E}{U} + 1 \tag{15}$$

Since E is much larger that U, it can be deducted that $\frac{2E+U}{EU}$ is much larger than $\frac{1}{E}$. Next, we will calculate this ratio numerically. As mentioned in §7.1, E is the equilibration constant, which is defined as 1.6×10^8, and U denotes the number of posting units assigned to this update. We take $U = 4480$, which is the same as that in §8.4. Thus, we can calculate the ratio numerically as follows.

$$\frac{\text{coefficient1}}{\text{coefficient2}} = \frac{2E}{U} + 1 = \frac{3.2\times 10^8}{4480} + 1 \approx 71,430 \tag{16}$$

As $\frac{2E+U}{EU}$ is considerably greater than $\frac{1}{E}$, it implies that the actual price change (ΔP_{n-1} in formula 14) will be significantly larger than the theoretically correct value (ΔP_{n-1} in formula 13). Thus, this implementation error will lead to a more rapid price increase.

We will also show experimentally that the difference will lead to a significant increase in the attack effects later. Specifically, the posting unit price can escalate to over 14,000 Gwei, representing a 700-fold increase from L1's gas price.

We further examine the cost of the optimization cost solution in its second step in practice, and the results are depicted in Figure 10. It can be seen that the average posting unit price in the second step is approximately 0.44 Gwei, which is very close to the theoretical value calculated in equation 11.

Given the exceedingly high posting unit price, it is hypothesized that only one transaction will occur on average every 10 seconds during the third stage of the attack. Moreover, the units of each transaction can be obtained through code instrumentation, with each transaction amounting to 1792 units. As a result, U is established as $U = 1792 * 2.5 = 4480$ for theoretical calculations. Figure 11 illustrates the effectiveness (i.e., attack impacts) caused by the cost optimization solution. The experimental results indicate a high degree of similarity with the theoretical value. Notably, the attack impacts' efficacy is significantly magnified by the implementation error in the pricing algorithm. More specifically, the posting unit price can escalate to over 14,000 Gwei, representing a 700-fold increase from the gas price on L1.

9 Disclosure and Mitigations

Vulnerability disclosure and ethics consideration. We disclosed all the vulnerabilities we found to Arbitrum via the Immunefi platform [28] and to Optimism via email. Arbitrum awarded corresponding bug bounties to us for all the vulnerabilities revealed by us. Both Arbitrum and Optmsim security teams followed our advice to fix all the vulnerabilities accordingly. Note that mitigating these vulnerabilities is non-trivial; both Arbitrum and Optimism have integrated new components and executed a hard fork of the L2 blockchain to fix these issues.

Mitigations on Arbitrum. To mitigate vulnerabilities under the manipulable delay attack, the overtime attack, and the QueueCut attack, Arbitrum developed a new component named data poster to submit batches to L1 in a streaming way [11], so that there is no need to wait for L1 confirmation, and the batch backlog issue is resolved. Hence, it is impossible for an attacker to bypass the limit of the submitter-only window to perform the corresponding double spending attacks. In the subsequent ArbOS version 10 hard fork, Arbitrum introduced further safeguards against the cost optimization solution of our delay attack. This mitigation involved utilizing ArbOS's storage variables to store the balance of the pool account [12], preventing them from being manipulated by attackers. To mitigate the vulnerabilities associated with Zip-Bomb attack, Arbitrum's batch submitter records the plaintext size of all transactions in each batch, and the batch submission operation is triggered when the plaintext size is close to the upper limit of the decompression size [10]. In this way, when each batch is decompressed in the L2 node, the decompressed data will not exceed the size limit.

Mitigations on Optimism. Several new mechanisms are introduced in Optimism's Bedrock hard fork to mitigate these vulnerabilities. First, a transaction pool is introduced to avoid handing new transactions in a first-come-first-serve manner, and users can pay priority fees to prioritize their transactions to be included in the next block [38]. In addition, the one-block-per-transaction strategy is changed to a fixed 2-second block interval. Furthermore, multiple batches are packaged into a channel, and a channel is split into several channel frames [39]. The batch submitter can post the channel frames to L1 in parallel. Therefore, after the Bedrock hard fork, the batch backlog issue was resolved, and it is impossible to launch the delay attack and the corresponding double spending attacks.

Mitigations on Cross-Chain Bridges. Upon reporting the vulnerabilities to Arbitrum, Arbitrum urged its cross-chain bridge partners to wait for L1 inclusion and not rely on sequencer confirmations when funds are leaving the system. As a result, several leading cross-chain bridge projects revised their codebase in accordance with this recommendation.

10 Related Work

Attacks against Ethereum. In the academic literature, a number of attacks against Ethereum at different system layers have been proposed, including P2P network [25, 26], Ethereum virtual machine [43, 49], transaction pool [32], and the RPC service [31]. For example, Heo [26] presented a practical partitioning attack named Gethlighting that isolates an Ethereum full node from the rest of the network for hours without having to eclipse all of the target's peer connections. Perez [43] presented the resource exhaustion attack on Ethereum, which exploits the significant inconsistencies in the pricing of the instructions. The authors also designed a genetic algorithm that generates contracts with throughput on average 100 times slower than typical contracts. Yang [49] developed a multi-transaction differential fuzzer named Fluffy and found 2 consensus bugs in the Geth Ethereum client. Li [32] proposed a low-cost denial-of-service attack against the Geth Ethereum client named DETER. An adversary can leverage the DETER attack to disable a remote Ethereum node's txpool and deny the critical downstream services. There are also several notable works focused on smart contract security [33, 47, 51].

Attacks against layer-2. Currently, the research on layer-2 security mainly focuses on the Bitcoin lighting network. Giulio [34] proposed the wormhole attack against the payment-channel networks (PCNs) which allows dishonest users to steal the payment fees from honest users along the path. This attack was acknowledged by the developers of the Bitcoin lighting network. In addition, the authors formally defined a new cryptographic primitive named anonymous multi-hop locks that can be used to design secure and privacy-preserving PCNs. Herrera-Joancomartí [27] proposed an attack to disclose the balance of a payment channel in the Bitcoin lighting network. The main idea of the attack is to perform multiple payments, ensuring that none of them is finalized to minimize the economic cost of the attack. Riard [44] proposed the time-dilation attack which dilates the time for victims to become aware of new blocks by isolating victims from the network and delaying block delivery. An attacker can leverage the time-dilation attack to steal funds from the victim's payment channels.

11 Conclusion

In this paper, we conduct an in-depth analysis of the optimistic rollup protocols and propose the manipulable delay attack to delay the confirmation of the blockchain state, and three double spending attacks. In addition, we also conducted extensive experiments to evaluate the costs and impacts of these attacks. All of the vulnerabilities found in this work were confirmed by corresponding official developers, and our research prevented billions of dollars in losses.

Acknowledgements

The authors thank the anonymous reviewers for their constructive comments. This work was partially supported by Hong Kong RGC Projects (No. PolyU15224121 and No. PolyU15231223).

References

[1] 2023. Inside Arbitrum Nitro. https://docs.arbitrum.io/inside-arbitrum-nitro/
[2] 2024. Arbitrum. https://arbitrum.io/
[3] 2024. Block numbers and time. https://docs.arbitrum.io/for-devs/concepts/differences-between-arbitrum-ethereum/block-numbers-and-time
[4] 2024. Optimism. https://www.optimism.io/
[5] 2024. The Sequencer and Censorship Resistance. https://docs.arbitrum.io/sequencer
[6] 2024. The state of the layer two ecosystem. https://l2beat.com/scaling/tvl
[7] Alexey Akhunov, Eli Ben Sasson, Tom Brand, Louis Guthmann, and Avihu Levy. 2019. EIP-2028: Transaction data gas cost reduction. https://eips.ethereum.org/EIPS/eip-2028
[8] Arbitrum 2022. Arbitrum repository. https://github.com/OffchainLabs
[9] Arbitrum 2022. Arbitrum transaction life cycle. https://docs.arbitrum.io/tx-lifecycle
[10] Arbitrum 2022. [Bug Fix] Add Max Decompressed Length to the Batch Poster. https://github.com/OffchainLabs/nitro/commit/ed149faf9732e1a626c502e1b05aa9f2eb41c9cc
[11] Arbitrum 2022. [Bug Fix] Add modular data poster to allow batch poster to queue up txs. https://github.com/OffchainLabs/nitro/pull/877
[12] Arbitrum 2022. [Bug Fix] ArbOS 10 Upgrade. https://github.com/OffchainLabs/nitro/commit/51f8bf7c101c1ce32477aab4d222bfe44d04597b
[13] Arbitrum 2022. L2 transaction fees on Arbitrum. https://docs.arbitrum.io/inside-arbitrum-nitro/#fees
[14] Arbitrum 2022. Maximum Batch Size in Arbitrum. https://github.com/OffchainLabs/nitro/blob/6dd0e3c2b1faccdb70f44514014c206f21c9745d/arbnode/batch_poster.go#82
[15] Arbitrum 2022. The Speed Limit Mechanism. https://docs.arbitrum.io/inside-arbitrum-nitro/#the-speed-limit
[16] Arbitrum 2022. Time Bound Mechanism in Arbitrum Codebase. https://github.com/OffchainLabs/nitro/blob/6dd0e3c2b1faccdb70f44514014c206f21c9745d/arbstate/inbox.go#L395-L399

[17] Arbitrum 2022. Two levels of transaction finality. https://docs.arbitrum.io/learn-more/faq#how-many-blocks-are-needed-for-a-transaction-to-be-confirmedfinalized-in-arbitrum.
[18] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *POST*.
[19] Lee Bousfield, Rachel Bousfield, Chris Buckland, Ben Burgess, Joshua Colvin, Edward W. Felten, Steven Goldfeder, Daniel Goldman, Huddleston Brade, Harry Kalodner, Frederico Arnaud Lacs, Harry Ng, Aman Sanghi, Tristan Wilson, Valeria Yermakova, and Tsahi Zidenberg. 2022. Arbitrum Nitro: A Second-Generation Optimistic Rollup. https://github.com/OffchainLabs/nitro/blob/master/docs/Nitro-whitepaper.pdf
[20] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. 2019. EIP-1559: Fee market change for ETH 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559
[21] Davide Crapis, Edward W Felten, and Akaki Mamageishvili. 2023. EIP-4844 Economics and Rollup Strategies. *arXiv* (2023).
[22] defillama 2023. Cross-Chain Bridge Statistical Data. https://defillama.com/bridges
[23] Github 2023. Go-Ethereum. https://github.com/ethereum/go-ethereum
[24] Hardhat 2023. Hardhat: Ethereum development environment for professionals. https://hardhat.org/
[25] Sebastian Henningsen, Daniel Teunis, Martin Florian, and Björn Scheuermann. 2019. Eclipsing ethereum peers with false friends. *arXiv preprint arXiv:1908.10141* (2019).
[26] Hwanjo Heo, Seungwon Woo, Taeung Yoon, Min Suk Kang, and Seungwon Shin. 2023. Partitioning Ethereum without Eclipsing It. In *Proceedings 2023 Network and Distributed System Security Symposium. NDSS*.
[27] Jordi Herrera-Joancomartí, Guillermo Navarro-Arribas, Alejandro Ranchal-Pedrosa, Cristina Pérez-Solà, and Joaquin Garcia-Alfaro. 2019. On the difficulty of hiding the balance of lightning network channels. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 602–612.
[28] Immunefi 2022. The Immunefi Platform. https://immunefi.com/
[29] L2BEAT 2023. Risks of Arbitrum One. https://l2beat.com/scaling/projects/arbitrum
[30] L2BEAT 2023. The Operator of Optimism. https://l2beat.com/scaling/projects/optimism#operator
[31] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service.. In *NDSS*.
[32] Kai Li, Yibo Wang, and Yuzhe Tang. 2021. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1667.
[33] Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. 2024. Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[34] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. 2018. Anonymous multi-hop locks for blockchain scalability and interoperability. *Cryptology ePrint Archive* (2018).
[35] OPTIMISM 2022. L2 transaction fees on Optimism. https://community.optimism.io/docs/developers/build/transaction-fees/
[36] OPTIMISM 2022. Maximum L1 Transaction Size in Optimism. https://github.com/ethereum-optimism/optimism/blob/0851fdb6df67a519b3a80087f48bd285c855786e/ops/envs/batch-submitter.env#L8
[37] OPTIMISM 2022. Optimism gas-oracle. https://github.com/ethereum-optimism/optimism/tree/0851fdb6df67a519b3a80087f48bd285c855786e/gas-oracle
[38] OPTIMISM 2023. How is Bedrock Different? https://community.optimism.io/docs/developers/bedrock/how-is-bedrock-different/
[39] OPTIMISM 2023. L2 Chain Derivation Specification. https://github.com/ethereum-optimism/optimism/blob/develop/specs/derivation.md
[40] Optimism 2023. Official docker file to run Optimism. https://github.com/ethereum-optimism/optimism/tree/0851fdb6df67a519b3a80087f48bd285c855786e/ops
[41] OPTIMISM 2023. Responding to gas price updates Optimism. https://community.optimism.io/docs/developers/build/transaction-fees/#responding-to-gas-price-updates
[42] OPTIMISM 2023. The L1 data fee on Optimism. https://community.optimism.io/docs/developers/build/transaction-fees/#the-l1-data-fee
[43] Daniel Perez and Benjamin Livshits. 2019. Broken metre: Attacking resource metering in EVM. *arXiv preprint arXiv:1909.07220* (2019).
[44] Antoine Riard and Gleb Naumenko. 2020. Time-dilation attacks on the lightning network. *arXiv preprint arXiv:2006.01418* (2020).
[45] David Salomon. 2007. *Data Compression: The Complete Reference* (4th ed.). Springer, New York, NY.
[46] Corwin Smith. 2023. OPTIMISTIC ROLLUPS. https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/
[47] Zhiyuan Sun, Xiapu Luo, and Yinqian Zhang. 2023. Panda: Security analysis of algorand smart contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1811–1828.
[48] Trail Of Bits 2022. Security Assessment of Nitro. https://github.com/OffchainLabs/nitro/blob/master/audits/Trail_Of_Bits_Nitro_10_2022.pdf
[49] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing.. In *OSDI*. 349–365.
[50] Wuqi Zhang, Lili Wei, Shuqing Li, Yepang Liu, and Shing-Chi Cheung. 2021. Ðarcher: Detecting on-chain-off-chain synchronization bugs in decentralized applications. In *FSE*.
[51] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. Deepinfer: Deep type inference from smart contract bytecode. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 745–757.

A Fraud-Proof Protocol

The main idea of optimistic rollup is that anyone can submit a state root to the state manager, and the state manager expects these state roots to be correct but does not provide a guarantee. If someone submits the wrong state root, it will be corrected through the fraud-proof protocol. As mentioned in §2.1, validators may disagree on the state root posted to L1. In other words, validators may disagree on the final state of the L2 blockchain after executing some transactions. The fraud-proof protocol is designed to resolve disputes. In other words, the protocol is optimistic because it advances the state of the L2 blockchain by letting any validator post state root to the state manager that is claimed to be correct and then giving others a chance to challenge that claim. If the challenge period passes and nobody has challenged the published state root, the state root is considered valid and correct.

There are basically two design choices for fraud-proof protocols, the first is an interactive proof based on binary search, and the other is the re-execution of transactions on L1. Arbitrum uses interactive fraud-proof, and Optimism initially adopted the method of re-executing disputed transactions on L1 and later switched to interactive fraud-proof as well. The gist of interactive proving is that the parties in dispute will engage in a back-and-forth protocol refereed by a L1 smart contract named challenge manager. The interactive protocol is based on the dissection of the dispute, and the L2 execution engine will run in a virtual machine in order to record the intermediate execution state. If one party claims that the execution of the disputed transaction includes N steps (one instruction executed at each step), it posts two claims of size N/2, which combine to yield the N-step claim, and the other party chooses one of the N/2-step claims to challenge. This procedure is repeated, dividing the dispute in half at each stage until they only disagree on a single step of execution. Finally, this instruction will be executed in L1, and the challenge manager will judge which party won the challenge based on the execution result.