## Security Analysis of Smart Contract Migration from Ethereum to Arbitrum

Xueyan Tang, Lingzhi Shi, Alan Lai, Yuying Du, Jing Deng, Jialu Fu and Jiayi Li

Salus Security, Beijing, 100020, China Email: yuying@salusec.io

When migrating smart contracts from one blockchain platform to another, there are potential security risks. This is because different blockchain platforms have different environments and characteristics for executing smart contracts. The focus of this paper is to study the security risks associated with the migration of smart contracts from Ethereum to Arbitrum. We collected relevant data and analyzed smart contract migration cases to explore the differences between Ethereum and Arbitrum in areas such as Arbitrum cross-chain messaging, block properties, contract address alias, and gas fees. From the 36 types of smart contract migration cases we identified, we selected 4 typical types of cases and summarized their security risks. The research shows that smart contracts deployed on Ethereum may face certain potential security risks during migration to Arbitrum, mainly due to issues inherent in public blockchain characteristics, such as outdated off-chain data obtained by the inactive sequencer, logic errors based on time, the permission check failed, Denial of Service(DOS) attacks. To mitigate these security risks, we proposed avoidance methods and provided considerations for users and developers to ensure a secure migration process. It's worth noting that this study is the first to conduct an in-depth analysis of the secure migration of smart contracts from Ethereum to Arbitrum.

Keywords: Blockchain, Ehtereum, Arbitrum, Smart contract, Migration
Received 00 December 2023

### 1. INTRODUCTION

Smart contracts are automated execution programs running on blockchain platforms. With the continuous development and application promotion of blockchain technology, smart contract migration has become a topic of considerable interest [1]. Smart contract migration refers to the process of transferring a contract from one blockchain platform to another. It holds significant value in the blockchain realm [2], enabling cross-chain interoperability among various blockchain platforms and facilitating the free flow of data and assets across different platforms [3]. Through migration, projects can leverage the technology of new blockchain platforms to enhance the performance, security, and scalability of their contracts. will conduct research focusing on the smart contract migrating from Ethereum [4] to Arbitrum [5].

However, smart contract cross-platform migration faces security risks. Different blockchain platforms may adopt varying protocols, standards, smart contract languages, and virtual machines. This could result in a smart contract without any apparent flaws, when deployed on one blockchain platform, encountering security issues when executed on another platform. Therefore, to better mitigate potential security risks and ensure smooth contract migration [6], it's necessary to thoroughly assess the characteristics of the destination chain and the underlying differences between the source and destination chains prior to the migration [7]. The primary objective of our study is to help developers understand the underlying differences between Ethereum [4] and Arbitrum [5], thereby enhancing the safety of smart contract migration.

The contributions of this study are as follows:

- 1. We identify security risks related to the runtime environment of smart contracts, highlight the importance of understanding the underlying differences between the source and target blockchains during smart contract migration for vulnerability identification.
- 2. Through a multifaceted analysis of differences, we delved into the key distinctions between Ethereum and Arbitrum. Our research revealed several unique

aspects, including Arbitrum Cross-Chain Messaging, Block Properties, Contract Address Alias, and Gas Fees.

- 3. Through multiple case studies, we summarize the potential risks that may arise during the migration of smart contracts from Ethereum to Arbitrum. These include outdated off-chain data obtained by the inactive sequencer, logic errors based on time, the permission check failed, DOS attack.
- 4. To address these issues, we provide relevant mitigation strategies, which are valuable for users and developers with smart contract migration needs. It is worth mentioning that this paper is the first in-depth research on the secure migration of smart contracts from Ethereum to Arbitrum.

### 2. BACKGROUND

### 2.1. Ethereum

Ethereum is a platform that integrates distributed ledger technology, cryptographic techniques, consensus algorithms, and smart contracts. Its design goal is to achieve decentralization, security, transparency, and programmability. As of October 2023, Ethereum is one of the most widely used blockchain platforms, with a market value of approximately \$167.46 billion [8]. It provides the foundation and innovative opportunities for smart contracts, decentralized finance, crowdfunding, and token issuance, and has a strong community and developer ecosystem.

Smart Contracts: Smart contracts are programming protocols used to automatically execute and verify contract terms. They are mainly used for executing and managing transactions and the functionalities of digital assets. This concept was first introduced to the blockchain field by Ethereum. Ethereum smart contracts are typically developed using the Solidity programming language.

Solidity Programming Language: Solidity is a Turing-complete programming language that allows developers to implement various complex functions and logic using powerful features such as loop control structures, conditional statements, and recursive calls. In addition, Solidity supports advanced language features such as function calls, modifiers, overloading, events, inheritance, and libraries, greatly enhancing the programmability and flexibility of contracts.

Smart Contract Deployment: Smart contracts are deployed on blockchain platforms such as Ethereum. After deployment, a smart contract is assigned a unique contract address, which is used to identify and locate instances of the contract on the blockchain. Successfully deployed smart contract code is converted into bytecode that can be executed on the EVM.

EVM: The EVM is a stack-based virtual machine with its own instruction set and memory model. It is a fully isolated runtime environment for executing

smart contracts and maintaining the consistency of the Ethereum platform. Its design goal is to provide a secure and reliable environment. The EVM uses a gas-based mechanism to limit the computational and storage resource consumption of contracts.

Gas Fee: Gas fee is used to measure the computational complexity and resource consumption of contract execution. Each operation and instruction has a predefined gas cost, and the actual consumption is billed accordingly during execution. The gas mechanism ensures that the execution of smart contracts does not consume resources without limits, preventing issues such as malicious code or infinite loops. Paying sufficient gas fees ensures the smooth execution of smart contracts and correct result returns.

#### 2.2. Arbitrum

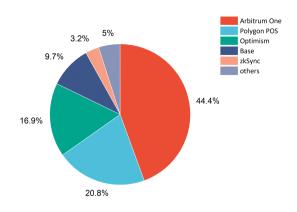
Arbitrum, developed by Offchain Labs, is a novel Ethereum L2 scaling solution that has gained significant attention and popularity on Ethereum community. Arbitrum supports fast smart contract transactions while reducing transaction costs. Arbitrum is highly compatible with the Ethereum Virtual Machine (EVM) [9]. Developers can write smart contracts using Ethereum tools like Solidity and easily migrate them to Arbitrum. Therefore, Arbitrum is also a popular choice for smart contract migration.

As of August 31, 2023, Arbitrum has captured 55.95% of the L2 market share [5]. There are two Arbitrum chains on Ethereum mainnet: Arbitrum One, an Arbitrum Rollup chain, and Nova, an AnyTrust chain. The data availability for Arbitrum One is on-chain (Ethereum mainnet), while the data availability for Nova is off-chain (Data Availability Committee).

The Arbitrum One Portal stores off-chain execution data on Ethereum mainnet. As of October 2023, a total of 576 projects have been deployed on Arbitrum One [10], such as Curve and OpenSea. As of October 2023, according to official statistics from Coingecko, the Total Value Locked (TVL) in Arbitrum One is over \$1.7 billion, making it the largest market share among all L2 chains today [11]. TVL is a measure of the total value of assets locked by users on decentralized finance (DeFi) platforms or applications. As shown in Figure 1, Arbitrum has the highest TVL share in L2.

### 2.3. The Migration of Smart Contracts from L1 to L2

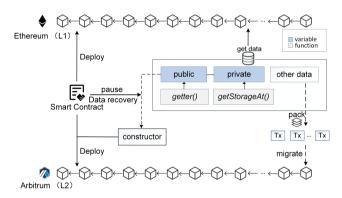
In this section, we primarily discuss the process of migrating smart contracts from Ethereum to L2, as well as the key features of these L2 related to smart contracts, such as the development language and runtime environment for smart contracts.



**FIGURE 1.** Distribution of TVL in L2. Arbitrum One has the largest share, followed by Polygon Pos and Optimism.

### 2.3.1. Smart Contract Migration Process

Smart contract migration mainly consists of two steps: recovering the data to be migrated and writing this data into a new contract deployed on the target blockchain. Figure 2 is Smart Contract Migration Process Diagram.



**FIGURE 2.** Smart Contract Migration Process Diagram. Migrating smart contracts deployed on Ethereum to Arbitrum, where Ethereum is the source blockchain for migration and Arbitrum is the target blockchain.

1. Data Recovery: The first step in smart contract migration is data recovery. In this step, we need to read data from specific blocks on the source blockchain for migration and use corresponding methods to recover the data based on the contract's data structure. For example, by calling the appropriate getter() function to retrieve public variables from the contract, relying on events, or using the getStorageAt() function to calculate the memory offset of private variables and retrieve their values from contract storage. However, please note that in order to increase the transparency of the migration and prevent attackers from exploiting users who are unaware of the migration, it is recommended to pause the contract's operation during the data recovery process.

2. Writing Data to New Contract: During smart contract migration, once the data to be migrated has been collected, the next step is to deploy and initialize the new contract on the target blockchain for migration. For simple variables, their values can be set through the contract's constructor() function. If the amount of data being migrated is large, the migration needs to be split into multiple transactions. For example, for ERC20 token contracts, the contract needs to be deployed on the target blockchain and the initial parameters set in the initial state. Then, users initiate transfer transactions in the old contract, and the new contract receives and records these transfer operations, moving the token balances from the old contract to the new one.

### 2.3.2. Differences Between L1 and L2

This paper mainly focuses on the case study of migrating smart contracts deployed on Ethereum to Arbitrum, revealing the security risks caused by migration. However, there are many smart contract migration solutions that are not limited to Arbitrum. L2 chains such as Polygon and StarkNet are also popular migration target blockchains. We explain the differences between L1 and L2 from the following two aspects:

- 1. Smart Contract Development Language: There are many programming languages for smart contract development, including Solidity [13], Vyper [14], Cairo [15], and more. Solidity is widely used for smart contract development on Ethereum and has a large developer community. Vyper is also an officially supported language for Ethereum. Most L2 blockchains also support Solidity, such as Arbitrum, Optimism [16]. Some L2 blockchains have their own smart contract development languages, such as Cairo for smart contracts on StarkNet [17]. Cairo is a low-level intermediate language used to describe the computation and state transition logic of StarkNet.
- 2. Smart Contract Runtime Environment: On Ethereum, the runtime environment for smart contracts based on Solidity and Vyper is the EVM. Many L2 solutions provide runtime environments for smart contracts that are EVM compatible, but the differences between them should not be overlooked. For example, Arbitrum's smart contracts [12] runtime environment simulates EVM execution through its custom Arbitrum Virtual Machine (AVM) or the WebAssembly-based ArbOS program. These execution environments aim to provide an execution environment that closely resembles the EVM. The operating environment for Polygon zkEVM [18] is the zkEVM developed by Polygon. Differences between Polygon zkEVM and EVM mainly include opcodes, precompiled contracts, and so on [19].

### 2.4. Vulnerabilities in smart contracts

Vulnerabilities in smart contracts refer to security weaknesses or flaws in the code of a smart contract, which can lead to unauthorized access, loss of funds, data tampering, or other adverse consequences. Some common smart contract vulnerabilities include weak random number generation, access control flaws, DOS attacks, and reentrancy.

Access Control Flaws occur when there are errors or inappropriate permission control mechanisms in a smart contract, allowing unauthorized or malicious users to execute operations or access data they shouldn't have access to.

DOS Attacks occur when malicious users exploit flaws or vulnerabilities in a contract to consume a large amount of resources or perform time-consuming operations, causing the contract to malfunction or block other users' operations.

Reentrancy refers to a vulnerability in which a smart contract fails to properly handle the invocation of an external contract, resulting in a reentrant call to a function within the current contract during the execution of the external contract. This type of vulnerability can lead to incorrect data state or financial loss within the contract.

Weak Randomness refers to the use of insecure or unreliable methods for generating random numbers in a smart contract, allowing attackers to predict or manipulate the outcome of random numbers and gain an unfair advantage.

The security of smart contracts extends beyond common vulnerabilities and also includes contract migration. Due to differences in implementation details, security features, and programming models between different platforms and runtime environments, these differences could potentially lead to security risks. To ensure the safe execution of smart contracts when migrating from Ethereum to Arbitrum, we investigate the underlying differences between these two environments, the features of Arbitrum, and the vulnerabilities in smart contracts caused by these differences. While these vulnerabilities also fall under common smart contract vulnerabilities, they cannot be easily identified without understanding the characteristics of the smart contract's runtime environment.

### 3. RELATED WORK

Smart contract migration is the process of transferring deployed smart contracts from one blockchain platform to another. Secure smart contract migration is an area of great interest. In fact, there have been some real-life cases that demonstrate the feasibility of smart contract migration. For example, Tank NFT (ERC721 Token Smart Contract) successfully migrated from Ethereum to Binance Smart Chain (BSC) [20]. BasketCoin also successfully migrated from Ethereum to BSC [21]. The

"The Sandbox" game project also migrated its smart contracts to the Polygon platform [22]. Blox NFT migrated from Ethereum to Arbitrum [23]. These cases demonstrate that smart contract migration is feasible and further highlight the importance and value of smart contract migration to reputable and promising blockchain platforms.

These cases indicate that smart contract migration is feasible, but special attention must be paid to security issues during the migration process. For example, the migration of the Uniswap governance system resulted in the inability to execute governance proposals. Due to poor communication among developers and misconfiguration of the system, the Uniswap governance system has encountered serious issues during the migration to Arbitrum, resulting in the inability to execute governance proposals on Arbitrum [24].

Furthermore, academic research provides us with valuable guidance. M. Westerkamp proposed a toolbox that enables the migration of smart contracts between EVM-compatible blockchains, using the migration of an ERC20 token contract as an example to demonstrate the effectiveness of the toolbox [25]. However, this research focuses on the verifiability of the code and state of the smart contract to be migrated, as well as the mechanism of smart contract migration using their proposed toolbox. They do not discuss the security issues of migration from the perspective of the smart contract. In contrast, our work focuses on security risks of the smart contract to be migrated. K. Shudo et al. emphasize the importance of application portability. They design middleware to facilitate the migration of applications between different blockchains to address the issue of mismatched incentives in public blockchains [26]. However, they also do not touch upon the portability and security of smart contract code and data.

### 4. METHODOLOGY

We have employed a mixed approach to study the security migration of smart contracts, which includes data collection and organization, case study, and expert involvement.

### 4.1. Data Collection

There are three main channels for data collection: relevant literature [27], official online resources [28], and smart contract source code cases [29]. These resources provide the latest research findings and industry dynamics, enabling us to understand cutting-edge knowledge in the field of smart contract security migration. We also categorized all the collected materials, creating directories based on different themes and keywords. Relevant official online resources and smart contract source codes were classified into corresponding categories.

DataPlatforms	Reference		
Google Scholar	[25] [26]		
Online Resources	[30] [31] [32] [33] [34]		
	[35] [36] [37] [38] [39]		
Smart Contract	[40] [41] [42] [43] [44]		
Vulnerability Cases	[45] [46] [47]		

**TABLE 1.** Source of Data

- Collection of Relevant Literature: During the process of collecting relevant literature, we used keywords such as "Smart Contract", "Migration" and "Security" to conduct literature searches on Google Scholar. We found that the number of relevant literature in the field of smart contract security migration is limited, which may be due to its relatively new and technologically complex nature. There are two papers [25] [26] that focus on discussing migration mechanisms by designing toolkits and middleware to facilitate the migration of smart contracts or applications. However, they do not provide explanations and verifications regarding the security of migrating smart contracts and applications.
- Collection of Official Online Resources: Although the amount of literature specifically focused on smart contract migration is limited, we made efforts to gather a large amount of official online resources to supplement the literature. These resources have been extremely helpful for Since our focus is on smart our research. contract migration from Ethereum to Arbitrum, we primarily relied on the official websites of Ethereum and Arbitrum for data collection [31, 32, 33, 34, 35, 36, 37]. In Particular, we found numerous relevant online resources on Arbitrum website, highlighting the differences between Arbitrum and Ethereum. Additionally, we collected data from the official websites of OpenZeppelin [48], Chainlink [49], and Solidity [13].
- Collection of Smart Contract Source Code: We collected cases of smart contract vulnerabilities from audit reports found on the official websites, including Code4rena [50], Cyfrin [51], Halborn [52], OpenZeppelin [48] and so on, encompassing a total of 36 types of smart contracts. We also referred to swcregistry (classification of smart contract vulnerability types), Ethereum Smart Contract Best Practices—Attacks, and the UNISWAP project. Table 1 displays some of our data sources.

### 4.2. Data Organization

Our team organized and categorized the collected materials and conducted case analysis. Based on the relevance of the content, the collected official online resources and smart contract source code will be categorized into 4 categories: Arbitrum cross chain messaging, block properties, address alias and gas fees. This classification method is mainly based on the supporting materials of the Offchain Labs team [31, 32, 33, 34, 35, 36, 37], who are the core leaders and development team of Arbitrum. It's worth noting that the description of the main differences between Arbitrum and Ethereum on the official Arbitrum website also primarily focuses on these 4 categories.

### 4.3. Case Study

Based on the organized materials, our team conducted an analysis of smart contract source code cases during the migration process from Ethereum to Arbitrum. Each member of our team has over two years of experience in smart contract development and auditing. They audited the collected source code cases and analyzed their security risks.

During our case analysis, we underwent the following three steps:

- 1. Analyze the underlying differences and characteristics between Ethereum and Arbitrum.
- 2. Based on the differences between Ethereum and Arbitrum's foundations, examine the smart contract source code to understand the contract's inheritance relationships, function call relationships, and various variables.
- 3. Combine automated vulnerability detection tools with manual auditing, and conduct group discussions on the audit results.
- 4. Synthesize case studies and foundational differences to summarize the potential security risks in smart contracts migrating from Ethereum to Arbitrum.

We organized a total of 36 types of related cases. These cases include smart contract source code with vulnerabilities, contract functionality, the causes of vulnerabilities, and Avoidance methods provided by our auditing experts. From the 36 types of smart contract migration cases we collected, we selected four typical cases. Among these, there is one case related to Arbitrum Cross Chain Messaging, two cases concerning Block Properties, one case on Contract Address Alias, and two cases about Gas Fees. These results are presented in the appendix.

### 4.4. Expert participation

To ensure the accuracy and comprehensiveness of our information collection, analysis, and organization, we have invited researchers from Offchain Labs to participate in our research. In the initial stages of the research, the experts provided us with targeted reference materials, which allowed us to conduct our research with a clear purpose. As our research reached milestone stages, due to physical distance, we communicated with the experts through online meetings and used document collaboration to guide our work.

The expertise and experience of these experts have provided us with valuable insights and recommendations [54], helping us gain a deeper understanding of the challenges and solutions involved in securely migrating smart contracts.

### 5. FINDINGS

Based on the academic literature, official online resources, and relevant smart contract vulnerability cases [55] we have collected, and under the guidance of experts, we have made the following findings: Although the Ethereum mainnet and the L2 scaling solution Arbitrum are both EVM-compatible blockchain platforms, there are still some noteworthy differences between them. These underlying differences can have an impact on the execution of smart contracts and may lead to security issues. In this section, we will explain the underlying differences between Ethereum and Arbitrum, as well as the impact these differences will have on smart contract migration from four aspects: Arbitrum Cross-Chain Messaging, Block Properties, Contract Address Alias and Gas Fees.

### 5.1. Arbitrum Cross-Chain Messaging

Arbitrum Cross-Chain Messaging is a mechanism for cross-chain interaction between Ethereum and Arbitrum. Through this mechanism, users can exchange messages and data between these two platforms to facilitate asset transfers, transaction execution, and other operations. Whether users are operating on Ethereum or Arbitrum, they can fully leverage the characteristics and ecosystems of each platform to achieve more efficient and secure cross-chain interactions.

Under the Arbitrum Cross-Chain Messaging mechanism, cross-chain message passing between Ethereum (referred to as L1) and Arbitrum (referred to as L2) is divided into two types: L2-to-L1 and L1-to-L2 transactions, as shown in Figure 3. Understanding these two transaction types and their mechanisms is crucial for ensuring smooth transaction execution and security. Different roles and contracts play important roles in these transaction types.

L2-to-L1 transactions refer to transfers or transaction operations from the Arbitrum to the Ethereum. In this transaction type, the Arbitrum sequencer plays a crucial role. The sequencer is a special node responsible for batching transactions on Arbitrum and submitting them to the Arbitrum on Ethereum. The role of the sequencer is to ensure that transactions on Arbitrum enter the Ethereum in the correct order, ensuring transaction correctness and security. Therefore, ensuring the smooth operation of the sequencer node is essential.

L1-to-L2 transactions are transfers or transaction operations from the Ethereum to the Arbitrum. This

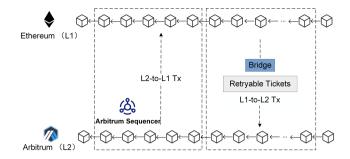


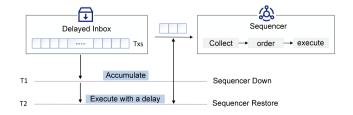
FIGURE 3. Transaction Classification Diagram. L2-to-L1 transactions require the involvement of the Arbitrum sequencer, while L1-to-L2 transactions are implemented through the bridge and retryable tickets.

type of transaction is facilitated through the Bridge mechanism. Users send assets from Ethereum to a specific Ethereum account, and then move these assets to the corresponding account on Arbitrum through the bridge. In L1-to-L2 transactions, retryable tickets also play a crucial role. Retryable tickets are special contracts used to ensure that transfers or transaction operations on Ethereum can be successfully executed on Arbitrum. If a transaction fails on Arbitrum, users can use the retryable tickets to retry the execution of that transaction. The preset gas fees in the L1-to-L2 message call process determine whether the transaction can be successfully executed.

In the following content, we will delve into the details of these transaction types and the potential security risks that may arise.

### 5.1.1. Sequencer in L2-to-L1 Message

The sequencer is a specifically designated full node in Arbitrum. Its main responsibility is to collect, order, and execute users' L2 transactions, providing immediate transaction results and receipts to users. Additionally, the sequencer periodically bundles multiple transactions and submits them to Ethereum to enhance the efficiency of the entire system. Regarding the Sequencer, the issue we discuss is the impact of its status on transaction execution.



**FIGURE 4.** Transaction Delay Execution Diagram. The sequencer's downtime results in transaction accumulation and delayed execution, compromising the real-time nature of transaction execution.

As shown in Figure 4, when the sequencer experiences downtime, transactions will accumulate in the delayed

inbox and cannot be executed immediately until the sequencer is restored. Once the sequencer is back online, it prioritizes processing all the old transactions in the delayed inbox before handling new transactions. However, these old transactions will still experience delayed execution.

In addition to sending L2 transactions to the sequencer, users or projects can also directly send L2 transactions to Ethereum's delayed inbox. delayed inbox is a smart contract deployed on Ethereum mainnet, protected by Ethereum's smart contract security mechanisms. However, it is worth noting that transactions sent directly to the delayed inbox by users will also experience delays until a specified time or block height, when they ultimately need to be executed by the sequencer. Thus, the execution time of these transactions will be delayed. Furthermore, when sending transactions to the delayed inbox, users need to manually set transaction parameters such as gas price and gas limit to ensure timely execution. Improper parameter settings may lead to transaction execution failures or the risk of overpaying transaction fees.

Issues and Risks: Considering that L2-to-L1 transactions rely on the sequencer to process transactions and bundle blocks, when migrating smart contract code from Ethereum to Arbitrum, it is crucial to consider the operational status of the sequencer, especially when the code involves obtaining real-time off-chain data. If the sequencer experiences downtime and reconnects, it may result in various issues, such as:

- 1. Incorrect transaction execution: If a contract relies on real-time off-chain data for executing transfers or other fund-related operations, delayed data can cause transaction execution delays or errors. This can lead to funds being transferred incorrectly or transactions not being executed as expected.
- 2. Asymmetric transaction conditions: Delayed offchain data may cause information asymmetry between smart contracts and external data sources, potentially leading to missed opportunities or unfair transaction conditions. This can result in certain participants obtaining unfair advantages, leading to financial losses for other participants.
- 3. Inaccurate prices or market data: If a smart contract relies on timely prices or market data for executing trades or making decisions, delayed or inaccurate data can lead to transactions being executed at inappropriate prices or market conditions, resulting in financial losses.

5.1.2. L1-to-L2 Messaging through Retryable Tickets The invocation of L1-to-L2 messaging can be done through retryable tickets, which is generally used for asset transfer. This process can be divided into two stages: submitting a ticket on L1 (calling the Inbox.createRetryableTicket() function); and executing the redeem operation on L2. The submission on L1 and the execution on L2 are asynchronous and

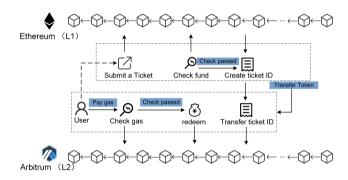
can be separated.

### 1. Submission process on L1

Submit a ticket on L1, which will generate a unique ticket ID. This process is asynchronous, meaning that users can continue with other operations after submitting the ticket without waiting for the execution on L2. Sufficient funds need to be provided for the ticket execution fees, and a fund check is also required. If the check passes, it indicates that the ticket creation was successful.

### 2. Redeem process on L2

After successfully creating the ticket, the redeem operation is performed on L2, transferring the assets from L1 to L2. Before execution, a gas check is required. If the check passes, the redeem operation is automatically executed. If the automatic redemption fails, the redeem operation needs to be performed manually. Figure 5 illustrates the L1-to-L2 messaging invocation.



**FIGURE 5.** Illustration of L1-to-L2 Messaging Invocation. Asset transfer is achieved based on ticket generation on L1 and the redeem operation on L2.

The redeem operation is automatically executed by the smart contract on L2. In this process, there are two possible scenarios:

Automatic redemption is successful: The submission fee is refunded to the L2 address specified by the user.

Automatic redemption fails: The submission fee is charged, and the remaining gas fee after execution is returned to the L2 specified address.

If automatic redemption fails, the ticket is temporarily stored in the L2 buffer for one week, and the submission fee is associated with this temporary storage. If automatic redemption is successful, it will not be stored in the buffer, and therefore, no submission fee needs to be paid.

**Issues and Risks:** In the L1-to-L2 messaging transfer, there is a possibility of failure during the submission process on L1 or the redeem process on L2, which may lead to adverse consequences:

1. When calling the createRetryableTicket() function on L1, if the funds fail the verification check, the transaction will be reverted, resulting in the loss of gas fees without any refund.

 After the automatic redemption fails on L2, a manual redeem operation is required. This may cause:

Overpayment of submission fees: In the case of automatic redemption success, the submission fee is refunded. However, in the case of manual redemption, the submission fee needs to be paid again, resulting in multiple fee payments.

Delay in transaction execution time: Manual redeem operation after automatic redeem failure may cause a delay in the execution time of the transaction, which could lead to losses such as price fluctuations.

Payment of L2 gas fees: Manual redeem operation also requires the payment of L2 gas fees to ensure proper execution of the transaction on L2.

3. If the ticket fails to redeem automatically and is temporarily stored in the buffer, it will be saved for seven days. If the ticket fails to redeem or the fee is not paid to continue the storage after the expiration, the carried messages and value (excluding the managed callvalue) may be lost and cannot be recovered.

### 5.2. Block Properties

Block properties refer to specific attributes or metadata that each block in Arbitrum possesses. These properties provide information about the block itself and the transactions contained within it. Examples of block properties include block height, timestamp, hash value, and the number of transactions included.

In Arbitrum and Ethereum, block properties generally refer to the block summary data stored in the block header. However, there may be some differences in the results obtained when retrieving Ethereum block properties. Let's take the block number and the block.timestamp as examples.

Table 2 shows the results obtained from Ethereum and Arbitrum when retrieving the Ethereum block number based on block.number. The Ethereum block time interval is approximately 15 seconds, while the Arbitrum sequencer requests an update of the L1 block number every minute. This means that the L1 block number obtained by Arbitrum will remain unchanged during this one-minute interval. In the 12:00-12:01 am interval, the block number obtained is consistently 1000.

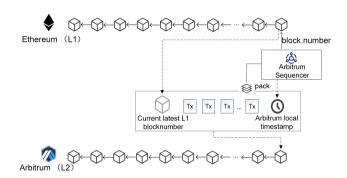
Results obtained from L1 Ethereum and L2 Arbitrum based on block.timestamp also differ. On Ethereum, the returned timestamp represents the current block's timestamp, while on Arbitrum, the timestamp recorded by the sequencer is obtained.

Below, we analyze the reasons for the aforementioned differences. As shown in Figure 6, when the sequencer packs transactions, it gathers various data. Three aspects of the data are relevant to our research: the current L1 block number obtained from Ethereum, which represents the latest block number on Ethereum;

Wall Clock time	L1	L2	
	block.number	block.number	
12:00:00am	1000	1000	
12:00:15am	1001	1000	
12:00:30am	1002	1000	
12:00:45am	1003	1000	
12:01:00am	1004	1004	
12:01:15am	1005	1005	

**TABLE 2.** Comparison of Wall Clock Times, L1-block.number, and L2-block.number. Data Source: https://docs.arbitrum.io/time

the timestamp recorded locally by the sequencer, which is used to assess the synchronization status of Arbitrum by comparing it to the l1 Timestamp obtained from Ethereum; and a certain number of transactions collected by the sequencer. If there are no issues, the sequencer will pack the L1 Block Number, timestamp, and these transactions together into a block.



**FIGURE 6.** Block Packaging Illustration. Among the data dependencies for block packaging, three aspects are relevant to our research: L1 Block Number, Local Timestamp, and Txs (transactions).

The method used by the sequencer to retrieve the local timestamp may return timestamps with low precision, and repeated retrieval within a very short time may result in the same timestamp being returned. In Arbitrum, the sequencer obtains the local timestamp to determine the block timestamp. Unlike Ethereum, Arbitrum does not impose a time difference between new blocks and their parent blocks. If the sequencer reads the time too frequently, it may result in different blocks having the same timestamp on Arbitrum.

Generally speaking, the acquisition of block numbers by smart contracts on Ethereum is continuous, while on Arbitrum (acquiring Ethereum's block numbers), it is discontinuous, with the value updated every minute. Because of this, if a smart contract on the Arbitrum uses a strict equality check (for example, currentNumber==customNumber), problems may arise. This is due to the fact that a specific customNumber value might not exist on the Arbitrum, so there's a certain probability that the conditions for the strict equality check cannot be met.

Issues and Risks: On Ethereum and Arbitrum, there are certain differences in the block properties obtained based on block.number and block.timestamp. These differences can lead to inconsistent behavior of smart contracts on Arbitrum and Ethereum, which may result in incorrect outcomes, security vulnerabilities, or other unpredictable behavior. Therefore, careful consideration of these differences and their potential security implications is necessary when migrating smart contracts from Ethereum to Arbitrum. Examples:

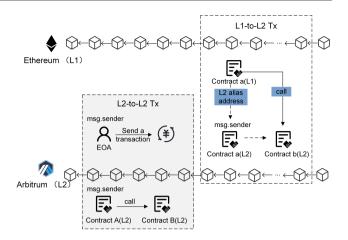
- 1. Inconsistent block number logic: Due to the different mechanism for updating L1 block numbers in Arbitrum compared to Ethereum, smart contracts may not be able to make accurate logical judgments based on block numbers. For example, if a contract relies on the state or events of a specific block, inconsistent results may occur on Arbitrum.
- 2. Inconsistent timestamp logic: Since the timestamp obtained in Arbitrum is recorded by the sequencer and not the timestamp of the current block, smart contracts may not be able to make accurate logical judgments based on timestamp. For example, if a contract needs to perform certain operations or restrictions based on the current time, inconsistent results may occur on Arbitrum.
- 3. Duplicate timestamp issue: It is possible for Arbitrum to have the same timestamp in different blocks, which can prevent contracts from accurately differentiating between timestamps of different blocks. This can impact time-related contract logic, such as restricting certain operations to be executed only once within a specific time period.

### 5.3. Contract Address Alias

Smart contract address alias refers to the new address assigned to a contract after migrating from Ethereum to Arbitrum. Setting an alias address for a smart contract is done to mitigate the risk of contract impersonation and ensure the security of cross-chain information transmission and interaction between Ethereum and Arbitrum.

When a smart contract is migrated from Ethereum to Arbitrum, it is assigned a new address, which results in a difference in 'msg.sender'. The 'msg.sender' is commonly used in smart contract code to return the address of the message sender. As shown in Figure 7, in L2-to-L2 transactions on Arbitrum, 'msg.sender' will return the address of the current message sender, consistent with the behavior on Ethereum. However, in L1-to-L2 transactions, 'msg.sender' will return the address alias of the L1 contract that triggered the message within L2.

The smart contract alias address calculation method is as follows: A represents the contract's address on L1, C is a constant, and f(A) represents the alias address of



**FIGURE 7.** Diagram of Retrieving 'msg.sender'. The 'msg.sender' returns the address of the message sender. However, it is important to note that on Arbitrum, the message sender may have an alias address.

the contract on L2.

$$f(A) = (A+C) \mod 2^{160} \tag{1}$$

The reason for setting an alias for the L1 contract address on Arbitrum is to avoid contract impersonation risks. Specifically, when a contract on Ethereum submits a transaction to Arbitrum, a problem arises regarding which sender address should be attached when the contract is running on Arbitrum. The simple solution is to use the L1 address of the sending contract, but there may be another contract on Arbitrum with the same address. If such a contract exists, the calling receiver on Arbitrum will not be able to distinguish between these two contracts, allowing one contract to impersonate another, which poses potential risks.

Issues and Risks: When migrating smart contracts from Ethereum to Arbitrum, it is important to be aware that if the contract involves L1-to-L2 message passing or comparing constant addresses with 'msg.sender', the alias address obtained from L1-to-L2 should be considered. If developers overlook the fact that the L1-to-L2 message retrieves an alias address, it may lead to the following security risks:

- 1. Failed permission checks: If a contract on Arbitrum performs permission checks by comparing 'msg.sender' with an expected address and does not take into account the alias address from L1-to-L2 messages, permission checks may fail. This means that unauthorized addresses may be granted access to the contract, resulting in potential security vulnerabilities.
- 2. Inability to modify contract owner: In some cases, a contract may need to modify its owner address. However, if the contract's owner address is obtained through L1-to-L2 messages and developers do not handle the alias correctly, the contract may be unable

to modify its owner. This can limit the contract's functionality and flexibility, as it may not be able to update its logic or configuration.

3. Potential contract impersonation risks: If a contract on Arbitrum uses 'msg.sender' for contract impersonation risk checks and does not consider the alias address, there may be a risk of contract impersonation. This means that one contract can impersonate another and perform unauthorized operations, potentially resulting in financial losses or other adverse effects.

#### 5.4. Gas Fees

Gas fees refer to the resource consumption required to execute smart contracts on blockchain platforms such as Ethereum and Arbitrum. Gas fees are necessary to compensate validators for their computational work and to prevent abuse or inefficient code execution. Each gas unit has a corresponding gas price, and miners determine the priority of transaction inclusion based on the gas price.

Gas fees consumed when executing smart contracts on Arbitrum is lower compared to Ethereum. This is because Arbitrum transactions are executed in batches, where multiple transactions are combined into a single batch for processing, reducing the fees and gas consumption per transaction. Additionally, Arbitrum moves a portion of the computation process off the Ethereum and processes it in an off-chain environment, further reducing the computational load and gas consumption on Ethereum.

We have obtained gas fees data for some transactions on Ethereum and Arbitrum, as shown in Table 3. For example, if a user performs a deposit operation on the Aave protocol deployed on Ethereum, it would cost \$4.02. However, if the same operation is performed on the Aave protocol deployed on Arbitrum One, it would only cost \$0.15, resulting in a 96% cost savings. The average gas consumption for the six transactions in the table on Arbitrum one is 95.5% lower than Ethereum.

Gas fees on Arbitrum consist of two aspects: firstly, the calldata of transaction invocation on Ethereum, which is intended to compensate the sequencer for the cost of publishing transactions on Ethereum, and secondly, the Arbitrum portion assessed and collected by ArbOS. The gas calculation formula is as follows:

$$Gaslimit = gasUsed_{L2} + \frac{calldataPrice_{L1} \times calldataSize_{L1}}{gasPrice_{L2}}$$
(3)

$$gasFees = Gaslimit \times gasPrice_{L2} \tag{4}$$

In the formula,  $gasUsed_{L2}$  stands for the amount of gas used for executing transactions on Arbitrum L2 platform.  $calldataPrice_{L1}$  stands for the price of L1

Tx	Arbitrum	Ethereum	Pct	Amount
Example	One		saved	saved
Aave	\$ 0.15	\$ 4.02	96%	\$3.87
Deposit				
EOA	\$ 0.09	\$ 0.65	87%	\$0.57
Transfer				
Opensea	\$ 0.20	\$ 5.55	96%	\$5.35
NFT Sale				
SushiSwap	\$ 0.08	\$ 2.53	97%	\$2.45
Swap				
Uniswap	\$ 0.08	\$ 3.97	98%	\$3.89
Swap				
Yearn	\$ 0.05	\$ 3.63	99%	\$3.58
Deposit				

**TABLE 3.** Gas fees Comparison between Arbitrum and Ethereum (Pct stands for Percentage). Data Source: https://gas.arbitrum.io/

calldata, indicating the cost of invoking transactions on Ethereum L1 platform.  $calldataSize_{L1}$  stands for the size of L1 calldata, indicating the number of bytes of data passed to the Arbitrum L2 sequencer.  $gasPrice_{L2}$  stands for the price of gas used on Arbitrum L2 platform. The calculated Gaslimit represents the total amount of gas used for transactions on Arbitrum L2 platform. The calculated gasFees represent the gas fees required for executing transactions on Arbitrum L2 platform.

Gas fees on Arbitrum are not fixed and can vary. This is because each block on Arbitrum has a fixed block space and gas limit, while the number of transactions within a block is uncertain. When there are more transactions within a block, the gas allocated per transaction decreases, and vice versa.

Gas fees required for transactions on Ethereum are relatively high, making it less likely for attackers to successfully launch a large number of small-value transactions to attack Ethereum. However, due to the lower gas fees on Arbitrum sometimes, attackers can carry out attacks involving a large number of small transactions. Therefore, it is still important to remain vigilant to ensure the security of our assets.

Issues and Risks: When gas fees on Arbitrum are relatively low, it can lead to DOS attacks.

- 1. Large-scale small-value transaction attacks: Due to the lower gas fees, attackers may exploit the Arbitrum to launch attacks involving a large number of small-value transactions. Such attacks can cause network congestion, transaction delays, and resource waste, negatively impacting the network's normal operation and user experience.
- 2. Resource exhaustion: Large-scale small-value transaction attacks can deplete network resources, including computational and storage resources. This can result in increased transaction processing times, transaction failures, or DOS issues, affecting users' normal transaction activities.

3. Exploitation of malicious contracts: Attackers may create malicious smart contracts to exploit the low gas fees and execute a large number of transactions for certain purposes, such as market manipulation, fraudulent activities, or other improper behaviors. This can cause significant losses and inconvenience to users and the ecosystem.

### 5.5. Summary of Issues

Based on the characteristics of Arbitrum and the differences between Arbitrum and Ethereum, we have summarized the issues that may arise when migrating smart contracts from Ethereum to Arbitrum, depending on what logic the smart contract contains.

### Outdated off-chain data obtained by the inactive sequencer:

Smart contracts that include logic to fetch off-chain real-time data and rely on the proper functioning of the sequencer need to be migrated from Ethereum to Arbitrum. If the sequencer is in an abnormal state and fetches outdated data, this can lead to incorrect transaction execution, asymmetric trading conditions, and inaccurate price or market data, resulting in financial losses and unfair trading conditions.

### Logic errors based on time:

Smart contracts containing the logic to retrieve block number and timestamp are being migrated from Ethereum to Arbitrum. This logic has resulted in inconsistencies in the behavior of the contract between Ethereum and Arbitrum, causing issues when dealing with time-related operations.

### The permission check failed:

On Arbitrum, ignoring address aliasing in L1-to-L2 transactions can lead to security issues during the migration process of smart contracts that involve permission checks logic. These issues may include permission check failures, inability to modify contract owners, and potential contract impersonation risks.

#### DOS attack:

When gas fees are low on Arbitrum, there may be security risks such as large-scale, small-value transaction attacks, resource exhaustion, and exploitation of malicious contracts.

To validate the aforementioned research findings, we conducted a case study analysis. The specific details can be found in the appendix.

### 6. RISK AVOIDANCE METHODS

### Outdated off-chain data obtained by the inactive sequencer:

When a smart contract running on Arbitrum needs to obtain real-time data, to avoid the issue of obtaining outdated data due to a Sequencer outage, Chainlink L2 Sequencer Uptime Feeds can be used. This involves checking whether the Sequencer is in normal operation

before relying on it to fetch data from off-chain. Only a Sequencer that is functioning normally can acquire timely off-chain data.

### Logic errors based on time:

To prevent unexpected errors in operations on Arbitrum that depend on block information like block numbers and timestamps, smart contracts running on Arbitrum should be mindful of potential delays when using block.number. Also, when using block.timestamp, it is important to be aware that the accuracy of the timestamps might not be high, which can lead to data inconsistencies.

In light of these issues, it's crucial to avoid using block information such as block.number and block.timestamp in scenarios where time sensitivity is critical. Furthermore, it's essential not to hardcode block numbers in smart contract code, which means not using specific block numbers for performing operations or querying specific blocks on the blockchain.

### The permission check failed:

If a smart contract's permission checks are based on the contract's address, it's crucial to pay special attention to the characteristic of address aliasing. Overlooking this aspect can lead to errors in permission checks. When migrating a smart contract from Ethereum to Arbitrum, it is necessary to set an alias for the smart contract. It's important to note that when msg.sender is retrieved, the returned address is the alias address of the contract on Arbitrum, not its address on Ethereum.

### DOS attack:

To reduce the likelihood of DoS attacks, for certain sensitive operations or users participating in a contract, introducing a fee or requiring a certain amount of collateral can be effective. Attackers would need to pay this fee to execute operations, thus increasing the cost of the attack.

In collateralized lending contracts, increasing the minimum deposit amount and introducing deposit time intervals can prevent frequent small deposits. Also, introducing a deposit threshold that requires the deposit amount to reach a certain level before updating the collateral ratio can be beneficial.

In crowdfunding projects, auction projects, or other contracts where batch refunds or other batch operations might occur, there may be arrays without explicit size limitations or constraints. Attackers could target a function or contract that processes arrays by adding a large amount of data to the array, leading to gas depletion during the loop process and causing the function to fail. To counter this, limiting the size of arrays can prevent attackers from adding excessive data that leads to too many iterations. Alternatively, using mappings instead of arrays and iterating over the keys of the mapping for operations can avoid the need for loops.

### 7. LIMITATIONS

#### 1. Data Collection:

We collected data including relevant literature, official online resources, and smart contract source code. We analyzed 6 typical cases of vulnerabilities or risks found in the collected data based on the underlying differences between Ethereum and Arbitrum. We also summarized the risks of migrating smart contracts from Ethereum to Arbitrum. However, due to limited data on smart contract migration, there may still be some potential risks that have not been disclosed. Additionally, it is important to note that there are no smart contracts that are absolutely free of vulnerabilities. We must be aware that during the migration process, we may encounter unknown risks.

### 2. Selection of Research Objects:

Our research focuses on Ethereum and Arbitrum. In the current blockchain technology landscape, there are various L1 and L2. L1 includes platforms like Hyperledger Fabric and EOS, while L2 includes solutions like Optimism and Polygon. The demand for smart contract migration exists within these L1 and L2. Different migration paths (i.e., from source blockchain to target blockchain) have underlying differences that introduce different security risks. We did not cover all possible migration paths. It is worth noting that we were fortunate to receive support from the Arbitrum Office Lab, which provided us with rich and authentic reference materials. These materials were crucial for our research, as they helped us conduct a more comprehensive and reliable research.

### 3. Migration Risks:

The smart contract vulnerabilities caused by migration originate from the contract's execution environ-These vulnerabilities do not exist before mi-Once the contract is migrated to another blockchain, the security risks appear. This is due to the underlying differences between the source and target blockchains, resulting in a change in the execution environment of the smart contract. Therefore, to effectively mitigate security issues caused by smart contract migration, auditors need to have specialized knowledge. Auditors should carefully analyze the blockchain protocols, virtual machines, and other related components of both the source and target blockchains. They need to understand the differences between different blockchains, including implementation details of the underlying protocols, features and limitations of smart contract languages, and behavioral differences of virtual machines, among others. Only with a deep understanding of these differences can auditors accurately assess the potential security risks during the contract migration process and provide relevant recommendations and solutions.

### 8. CONCLUSIONS AND FUTURE WORK

There are security risks involved in migrating smart contracts deployed on Ethereum to Arbitrum, which we have studied. In this research, we analyzed various security issues related to smart contract migration, including the features of Arbitrum, the differences between Arbitrum and Ethereum, and multiple vulnerable smart contract cases such as outdated off-chain data obtained by the inactive sequencer, logic errors based on time, the permission check failed, DOS attack. Furthermore, we provided we proposed avoidance methods and provided considerations for these potential security risks.

This paper primarily focuses on security risks during the migration process of smart contracts from Ethereum to Arbitrum. However, we must also acknowledge that apart from the smart contract migration issues discussed in this paper, there are other security risks. In fact, the security of smart contract migration is an evolving field with new challenges and potential risks constantly emerging. The security issues in smart contract migration may present even more challenges and risks when migrating between different blockchains. Whether it is migrating from Ethereum to Arbitrum, Ethereum to other Ethereum L2 scaling solutions (such as Optimism or Starknet), or even between Ethereum and other homomorphic chains (like Binance Smart Chain or Polygon), there are potential security risks involved.

Therefore, achieving secure smart contract migration requires continuous effort and exploration. The security issues in smart contract migration need to be continuously monitored, researched, and addressed. We recommend users carefully evaluate the characteristics, security, and feasibility of the migration target blockchain, as well as the underlying differences between the source and target blockchains, before proceeding with smart contract migration. It is also advisable to engage in discussions and consultations with relevant technical experts and communities to ensure a smooth migration process that aligns with the project's requirements.

### ACKNOWLEDGEMENTS

Thank you to Mehdi Salehi and ChrisCo from Offchain Labs for providing advice, reference materials, and conference support.

### REFERENCES

- [1] Fynn, Enrique, Bessani, Alysson, and Pedone, Fernando (2020) Smart contracts on the move. In: Proceedings of the 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, pp. 233–244.
- [2] Hafid, Abdelatif, Hafid, Abdelhakim Senhaji, and Samih, Mustapha (2020) Scaling blockchains: A comprehensive survey. IEEE Access, 8, 125244–125262.
- [3] Khan, Sajjad, Amin, Muhammad Bilal, Azar, Ahmad Taher, and Aslam, Sheraz (2021) Towards interoperable blockchains: A survey on the role of smart contracts in

- blockchain interoperability. IEEE Access, 9, 116672–116691.
- [4] Arbitrum (2023) Welcome to Ethereum. https://ethereum.org/ (Date last accessed: August 31, 2023).
- [5] Arbitrum (2023) Take it to the next layer with Nitro: Making Ethereum more inclusive and sustainable. ht tps://arbitrum.io/ (Date last accessed: August 31, 2023).
- [6] Neiheiser, Ray, Inácio, Gustavo, Rech, Luciana, Montez, Carlos, Matos, Miguel, and Rodrigues, Luís (2023) Practical Limitations of Ethereum's Layer-2. IEEE Access, 11, 8651–8662.
- [7] Roşca, Ionuţ, Butnaru, Alexandra-Ina, and Simion, Emil (2023) Security of Ethereum Layer 2s. Cryptology ePrint Archive.
- [8] CoinMarketCap (2023) Today's Cryptocurrency Prices by Market Cap. https://coinmarketcap.com/ (Date last accessed: August 31, 2023).
- [9] Kalodner, Harry, Goldfeder, Steven, Chen, Xiaoqi, Weinberg, S Matthew, and Felten, Edward W (2018) Arbitrum: Scalable, private smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 1353–1370.
- [10] Arbitrum (2023) Arbitrum Portal. https://port al.arbitrum.io/?chains=arbitrum-one (Date last accessed: August 31, 2023).
- [11] CoinGecko (2023) Top Layer 2 Chains by Total Value Locked (TVL). https://www.coingecko.com/en/chains/layer-2 (Date last accessed: August 31, 2023).
- [12] Arbitrum (2023) Why Nitro? https://docs.arbitrum.io/why-nitro (Date last accessed: August 31, 2023).
- [13] Solidity () Solidity. https://soliditylang.org/ (Date last accessed: August 31, 2023).
- [14] Vyper (2023) What is vyper? https://docs.vyperla ng.org/en/stable/ (Date last accessed: August 31, 2023).
- [15] Cairo (2023) The Cairo Programming Language https://book.cairo-lang.org/zh-cn/index.html (Date last accessed: August 31, 2023).
- [16] Optimism (2023) Ethereum, scaled. https://www.optimism.io/ (Date last accessed: August 31, 2023).
- [17] Starknet (2023) Welcome to Starknet. https://www.starknet.io/en (Date last accessed: August 31, 2023).
- [18] Polygon zkEVM (2023) Bring Ethereum to everyone. https://polygon.technology/polygon-zkevm (Date last accessed: August 31, 2023).
- [19] Polygon (2023) Differences between EVM and zkEVM. https://wiki.polygon.technology/docs/zkevm/p rotocol/evm-diff/ (Date last accessed: August 31, 2023).
- [20] Musson, Melvin (2023) CryptoTanks: Tank NFTs Smart Contract Migration. https: //medium.com/@cryptotanksio/cryptotanks-tan k-nfts-smart-contract-migration-60ace6c93663 (Date last accessed: August 31, 2023).
- [21] BasketCoin (2021) Migration to Binance Smart Chain (BSC). https://medium.com/@basketcoin/migration-to-binance-smart-chain-bsc-e42dd66cfe97 (Date last accessed: August 31, 2023).
- [22] The Sandbox (2022) The Sandbox is deploying on Polygon. https://medium.com/sandbox-game/the-s

- andbox-is-deploying-on-polygon-c44e026afeb2 (Date last accessed: August 31, 2023).
- [23] OpenBlox (2022) Migration guide: How to move your Blox from Ethereum to Arbitrum. https://medium.com/openblox/migration-guide-how-to-move-your-blox-from-ethereum-to-arbitrum-b736970830a3 (Date last accessed: August 31, 2023).
- [24] Fix the Cross Chain Messaging Bridge on Arbitrum (2022) https://gov.uniswap.org/t/temperature-check-fix-the-cross-chain-messaging-bridge-on-arbitrum/18073 (Date last accessed: August 31, 2023).
- [25] Westerkamp, Martin (2019) Verifiable smart contract portability. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), IEEE, pp. 1– 9.
- [26] Shudo, Kazuyuki, Kanda, Reiki, and Saito, Kenji (2018) Towards application portability on blockchains. In: 2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN), IEEE, pp. 51–55.
- [27] Papaioannou, Diana, Sutton, Anthea, and Booth, Andrew (2016) Systematic approaches to a successful literature review. Systematic approaches to a successful literature review, SAGE Publications Ltd, pp. 1–336.
- [28] Granello, Darcy Haag and Wheaton, Joe E (2004) Online data collection: Strategies for research. Journal of Counseling & Development, 82(4), 387–393.
- [29] Rashid, Yasir, Rashid, Ammar, Warraich, Muhammad Akib, Sabir, Sana Sameen, and Waseem, Ansar (2019) Case study method: A step-by-step guide for business researchers. International journal of qualitative methods, 18, 1609406919862424.
- [30] Arbitrum (2023) A Gentle Introduction to Arbitrum. https://docs.arbitrum.io/intro/ (Date last accessed: August 31, 2023).
- [31] Arbitrum (2023) Arbitrum Address Aliasing. https://docs.arbitrum.io/arbos/l1-to-l2-mes saging#address-aliasing (Date last accessed: August 31, 2023).
- [32] Arbitrum (2023) L2 to L1 Messaging. https://docs.arbitrum.io/arbos/12-to-11-messaging (Date last accessed: August 31, 2023).
- [33] Arbitrum (2023) ArbOs. https://docs.arbitrum.io/arbos/ (Date last accessed: August 31, 2023).
- [34] Arbitrum (2023) ArbOs Gas. https://docs.arbitrum. io/arbos/gas (Date last accessed: August 31, 2023).
- [35] Arbitrum (2023) ArbOs L1 Pricing. https://docs.arbitrum.io/arbos/11-pricing (Date last accessed: August 31, 2023).
- [36] Arbitrum (2023) The Sequencer. https://docs.arbit rum.io/sequencer (Date last accessed: August 31, 2023).
- [37] Arbitrum (2023) Transaction Lifecycle in Arbitrum. https://docs.arbitrum.io/tx-lifecycle (Date last accessed: August 31, 2023).
- [38] Ethereum (2023) Ethereum learn. https://ethereum.org/en/learn/ (Date last accessed: August 31, 2023).
- [39] ChainLink (2023) ChainLink l2-sequencer-feeds. https://docs.chain.link/data-feeds/12-sequenc er-feeds (Date last accessed: August 31, 2023).

- [40] Evert0x (2023) GLPOracle. https://github.com/s herlock-audit/2023-01-sentiment/blob/main/o racle/src/gmx/GLPOracle.sol (Date last accessed: August 31, 2023).
- [41] r0ohafza (2023) GLPOracle.t.sol. https://github.com/sentimentxyz/oracle/blob/815233add2d23a7e2a2c5136504537b234a65c47/src/tests/GLPOracle.t.sol (Date last accessed: August 31, 2023).
- [42] rOohafza (2023) GLPOracleUpdate.sol. https://github.com/sentimentxyz/oracle/blob/main/src/gmx/GLPOracle.sol (Date last accessed: August 31, 2023).
- [43] TriHaz (2022) Trading.sol. https://github.com/cod e-423n4/2022-12-tigris/blob/588c84b7bb354d20c bca6034544c4faa46e6a80e/contracts/Trading.sol #L857-L868 (Date last accessed: August 31, 2023).
- [44] tintinweb (2022) MarginFactory.sol. https://github.com/tintinweb/smart-contract-sanctuary-arbitrum/blob/662d22a0f98c6a0c8ef23e43ac6d6a3eac5968da/contracts (Date last accessed: August 31, 2023).
- [45] Uniswap (2021) UniswapV3Factory. https: //arbiscan.io/address/0x1F98431c8aD98523631A E4a59f267346ea31F984#contracts (Date last accessed: August 31, 2023).
- [46] DavidBDiligence (2022) DOS. https://github.com /Consensys/smart-contract-best-practices/blob/ master/docs/attacks/denial-of-service.md (Date last accessed: August 31, 2023).
- [47] Sherlock-admin (2023) Pool. https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L216-L263 (Date last accessed: August 31, 2023).
- [48] OpenZeppelin (2023) OpenZeppelin. https://www.openzeppelin.com/ (Date last accessed: August 31, 2023).
- [49] Chainlink (2023) Chainlink. https://chain.link/ (Date last accessed: August 31, 2023).
- [50] Code4rena () code4rena. https://code4rena.com/ (Date last accessed: August 31, 2023).
- [51] Code4rena () code4rena. https://www.cyfrin.io/ (Date last accessed: August 31, 2023).
- [52] Code4rena () code4rena. https://www.halborn.com/ (Date last accessed: August 31, 2023).
- [53] Ciotta, V, Mariniello, G, Asprone, D, Botta, A, and Manfredi, G (2021) Integration of blockchains and smart contracts into construction information flows: Proof-of-concept. Automation in Construction, 132, 103925.
- [54] Karasev, OI and Mukanina, EI (2019) Expert assessment method in foresight studies. Statistics and Economics, 16(4), 4–13.
- [55] Tantikul, Phitchayaphong and Ngamsuriyaroj, Sudsanguan (2020) Exploring Vulnerabilities in Solidity Smart Contract. ICISSP, pp. 317–324.

### APPENDIX A. ARBITRUM CROSS-CHAIN MESSAGING

# Appendix A.1. Issue: Outdated off-chain data was obtained by the inactive sequencer

Case: When obtaining off-chain data from an oracle, real-time requirements are crucial. If the sequencer goes down and transactions cannot be immediately executed, the contract will not be able to return accurate and real-time off-chain data.

Taking the GLPOracle.sol [40] contract as an example, we first introduce the main functionality of this contract. The getPrice() function is used to calculate the GLP/USD price. The constructor of the contract accepts two parameters: \_manager and \_ethFeed, representing the addresses of the GLPManager contract and the Ethereum price data source, respectively. The contract includes two functions: getPrice() and getEthPrice().

The getEthPrice() function retrieves Ethereum price by calling the latest ethUsdPriceFeed.latestRoundData() and performs validations for data expiration and negativity. ethUsdPriceFeed is an instance of an oracle contract used to provide external data services, and latestRoundData() is a public function of this contract that returns the latest ETH/USD price.

The getPrice() function first calls manager.getPrice(false) to obtain the current price of the GLP token, and then calls getEthPrice() to retrieve the latest ETH/USD price. The GLP/USD price is calculated as

### $\frac{\texttt{manager.getPrice(false)}}{\texttt{getEthPrice()} \times 10^4}$

where the GLP token price is divided by the ETH/USD price and then multiplied by 10<sup>4</sup>. This is done to convert the price unit of ETH to USD in order to obtain the price unit of GLP.

```
// SPDX-License-Identifier: MIT
  pragma solidity ^0.8.17;
  import {Errors} from "../utils/Errors.sol";
  import {IOracle} from "../core/IOracle.sol";
  import {IGLPManager} from "./IGLPManager.sol";
  import {AggregatorV3Interface} from
      "../chainlink/AggregatorV3Interface.sol";
  contract GLPOracle is IOracle {
      /// @notice address of gmx manager
      IGLPManager public immutable manager;
11
      /// @notice ETH USD Chainlink price feed
12
      AggregatorV3Interface immutable
          ethUsdPriceFeed;
14
          Onotice Contract constructor
16
          @param _manager address of gmx vault
          @param _ethFeed address of eth usdc
              chainlink feed
```

```
19
20
       constructor(IGLPManager _manager,
           AggregatorV3Interface _ethFeed) {
           manager = _manager;
21
           ethUsdPriceFeed = _ethFeed;
22
      7
23
24
      /// @inheritdoc IOracle
25
       function getPrice(address) external view
26
           returns (uint) {
           return manager.getPrice(false) /
27
               (getEthPrice() * 1e4);
28
       function getEthPrice() internal view
30
           returns (uint) {
           (, int answer,, uint updatedAt,) =
31
               ethUsdPriceFeed.latestRoundData();
           if (block.timestamp - updatedAt >=
               86400)
               revert
33
                   Errors.StalePrice(address(0),
                   address(ethUsdPriceFeed));
           if (answer <= 0)
               revert
35
                   Errors.NegativePrice(address(0),
                   address(ethUsdPriceFeed));
           return uint(answer):
36
37
      }
38 }
```

Security Risks: There is a potential vulnerability when executing this contract on Arbitrum. When the contract reaches the statement ethUsdPriceFeed.latestRoundData(), it needs to access the oracle through the sequencer to obtain off-chain price data. If the sequencer goes down at this moment, the statement cannot be executed immediately. When the sequencer reconnects and executes this statement, it will retrieve outdated price data, which may be higher or lower than the actual price. Attackers can exploit the price difference between the actual price and the outdated price for profit.

Exploitation of this vulnerability: Assuming a user borrows using GLP tokens as collateral, if the sequencer goes down and the reconnected sequencer retrieves an outdated price higher than the actual price, the user can obtain better borrowing conditions. If the outdated price is lower than the actual price, the user can avoid liquidation.

The above vulnerability has been tested and confirmed to exist. The test code can be found at [41].

### Risk Avoidance Methods:

To address the aforementioned vulnerability, we suggest querying the Chainlink L2 Sequencer Uptime Feeds to determine the operational status of the sequencer [42].

```
9 contract GLPOracle is IOracle {
      /// @notice address of gmx manager
11
      IGLPManager public immutable manager;
      /// @notice ETH USD Chainlink price feed
      AggregatorV3Interface immutable
           ethUsdPriceFeed;
       /// @notice L2 Sequencer feed
14
      AggregatorV3Interface immutable sequencer;
      /// @notice L2 Sequencer grace period
16
      uint256 private constant GRACE_PERIOD_TIME
17
          = 3600:
18
19
       constructor(IGLPManager _manager,
          AggregatorV3Interface _ethFeed,
          AggregatorV3Interface _sequencer) {
          manager = _manager;
20
           ethUsdPriceFeed = _ethFeed;
21
           sequencer = _sequencer;
22
23
24
      function getPrice(address) external view
          returns (uint256) {
           if (!isSequencerActive()) revert
26
               Errors.L2SequencerUnavailable();
          return manager.getPrice(false) /
               (getEthPrice() * 1e4);
      }
28
29
      function getEthPrice() internal view
          returns (uint256) {
           (, int256 answer,, uint256 updatedAt,)
               ethUsdPriceFeed.latestRoundData();
           if (block.timestamp - updatedAt >=
               86400) {
               revert
                   Errors.StalePrice(address(0),
                   address(ethUsdPriceFeed));
          }
34
           if (answer <= 0) {
               revert.
                   Errors.NegativePrice(address(0),
                   address(ethUsdPriceFeed));
          }
          return uint256 (answer);
38
39
40
      function isSequencerActive() internal view
          returns (bool) {
           (, int256 answer, uint256 startedAt,,)
42
               = sequencer.latestRoundData();
           if (block.timestamp - startedAt <=</pre>
               GRACE_PERIOD_TIME || answer == 1) {
44
               return false;
45
          return true;
47
      }
```

In the getPrice() function, the first step is to call the isSequencerActive() function to check if the sequencer is in a normal operational state. If it is not normal, the function will revert. Only when the sequencer is running normally, the code to retrieve off-chain price data and calculate the GLP price will be executed. This logic aligns with the main functionality of the GLPOracle.sol [40] contract mentioned above.

The isSequencerActive() function retrieves the status of the sequencer by calling sequencer.latestRoundData(). The sequencer

is an instance of an oracle contract used to provide external data services, and it primarily uses the sequencer uptime feed proxy address for configuration.latestRoundData() is a public function in that contract, and it returns the status of the sequencer. If it is running normally, the function returns true; otherwise, it returns false.

#### APPENDIX B. BLOCK PROPERTIES

### Appendix B.1. Issue: Logic errors based on time

Case 1: The smart contract project for financial derivatives trading relies on block.number to calculate time intervals. Specifically, the \_checkDelay() [43]function implements a locking mechanism to check if there is enough time between opening and closing positions. This is done to prevent profiting from opening and closing positions with two different prices in the same transaction within the valid signature pool.

```
function _checkDelay(uint _id, bool _type)
       internal {
       unchecked {
           Delay memory _delay =
3
                blockDelayPassed[_id];
           if (_delay.actionType == _type) {
    blockDelayPassed[_id].delay =
                     block.number + blockDelay;
           } else {
                if (block.number < _delay.delay)</pre>
                     revert("0");
                blockDelayPassed[_id].delay =
                     block.number + blockDelay;
                blockDelayPassed[_id].actionType =
                     _type;
           }
       }
12 }
```

Security Risks: While this structure works fine on Ethereum, it has issues when used on Arbitrum. The sequencer returns the most recently synchronized L1 block number based on block.number every minute. This one-minute time interval can be exploited. Users can open a position before the synchronization occurs (e.g., at 12:00:45 am, L2 obtains L1 block number 1000), and then close it in the next block (e.g., at 12:01 am, L2 obtains number 1004). It may seem like there have been 5 L1 blocks (60/12) since the last transaction, but in reality, there haven't been enough L1 blocks delayed to bypass the locking protection.

Malicious traders can exploit this by continuously updating the block delay in \_checkDelay() and increasing the stop-loss threshold, enabling risk-free trading. This is a problem inherent to L1 itself, but if it occurs on Arbitrum, the impact will be amplified as malicious traders can modify the time delay for closing positions without going through the blockDelay.

Risk Avoidance Methods: Replace 'block.number' with 'block.timestamp'. This way, the code can ensure the correct time delay on both

Ethereum and Arbitrum, since 'block.timestamp' always increases according to real time, not based on the block production rate. This can prevent abuse on Arbitrum.

Case 2: In an Arbitrum smart contract, there are check statements to restrict users from performing multiple operations within a single block. For example, the time check statements in the openPosition() and closePosition() functions [44] are as follows:

```
// Restrict users from performing multiple
    operations within a single block
traderLatestOperation[trader] !=
    block.number,
    "ONE_BLOCK_TWICE_OPERATION"
);
```

Security Risks: Arbitrum updates 'block.number' every minute. However, in reality, several blocks may have passed on L1, but the obtained L1 block number on Arbitrum remains unchanged. This causes a delay of up to one minute in the operations performed in the example code.

Risk Avoidance Methods: Use 'block.timestamp' in place of 'block.number'. The 'block.timestamp' refers to the timestamp of the current block, which is set by the miner producing the block and is generally close to the actual time the block was produced. This method can avoid issues caused by delays in the update of 'block.number'.

However, it is important to note that 'block.timestamp' is not entirely reliable, as miners have a degree of freedom in setting this value (usually within certain limits). This could lead to potential security risks. In most cases, these risks are acceptable, but in scenarios requiring very high security, 'block.timestamp' should be used with caution.

### APPENDIX C. CONTRACT ADDRESS ALIAS

### Appendix C.1. Issue: The permission check failed

Case: Uniswap Labs did not consider Address Alias when deploying to Arbitrum.

Security Risks: When deploying the Uniswap v3 Factory [45]to Arbitrum, the owner of the Factory contract is set to the original address of the Timelock contract on Ethereum using the setOwner() function. However, during the L1-to-L2 message call, the msg.sender obtained is the alias address of the Timelock contract on Arbitrum. As a result, the permission check cannot be passed. Additionally, as

this alias address is an externally owned account (EOA) and no one possesses the private key for this address, the owner cannot be modified, and functions that require owner permissions cannot be executed on Arbitrum for the Factory contract.

### Risk Avoidance Methods:

To address this issue, Arbitrum temporarily disabled address alias for the Timelock contract. In the Inbox contract, there is a specific method created for Uni called uniswapCreateRetryableTicket. In the absence of address alias, the Uniswap Factory contract sends a cross-chain message from the Ethereum Timelock contract to invoke the setOwner() function on Arbitrum Uniswap Factory, setting the owner to the alias address Ox2BAD8182C09F50c8318d769245beA52C32Be46CD of the Ethereum Timelock contract. This ensures compliance with the permission check in the Factory contract.

### APPENDIX D. GAS FEES

### Appendix D.1. Issue: DOS attack

Case 1: Each block has an upper limit on the amount of gas that can be consumed. If the gas spent exceeds this limit, the transaction will fail. In a contract, there may be arrays without explicit size restrictions or constraints. An attacker may intentionally add a large amount of data to an array in a function or contract that processes the array, causing the gas to be exhausted during the loop, rendering the function unable to execute successfully.

This situation is prone to occur in crowdfunding projects, auction projects, or other contract projects that may involve batch refunds or other batch operations. For example, an attacker may add a large number of addresses to the contract, each requiring a very small refund. When the project contract attempts to refund by iterating through the array, the loop count becomes enormous due to the attacker's addition of a large number of addresses, and the gas cost for this transaction may ultimately exceed the gas limit, resulting in the inability to refund. Since gas prices are lower on Arbitrum, the attacker's cost is reduced, making such attacks more likely to occur.

In the code snippet below [46], a for loop is used to iterate through each address in the Addresses array.

Security Risks: In each iteration, it uses a require statement to invoke the transfer() function of the

Token contract, transferring the funds stored in the contract to the corresponding addresses. If the attacker adds a large number of addresses to the Addresses array, and each address requires a very small refund amount, the loop in the refundAll() function will execute a significant number of iterations. When the loop count exceeds the gas limit of the block, the transaction will fail, resulting in the inability to successfully execute the refunds. In this scenario, the attacker can prevent the refund operation by depleting the gas, thereby affecting the normal operation of the contract.

#### Risk Avoidance Method 1:

To address this issue, several approaches can be considered:

1.Limiting the array size: Adding a function or modifier to the contract that limits the size of the Addresses array can prevent attackers from adding an excessive number of addresses, thereby reducing the number of iterations in the loop.

2.Batch processing: Dividing the refund operation into multiple loops, with each loop processing a certain number of addresses, can prevent excessive data processing in a single iteration and reduce the number of loops.

3. Using mappings instead of arrays: Consider using mappings instead of arrays to store addresses and refund amounts. Mappings have no size limitation, and refund operations can be handled by iterating through the keys of the mapping without the need for a loop.

4. Using optimized algorithms: Employ more efficient algorithms to handle refund operations, reducing the number of iterations and gas consumption. For example, binary search or other efficient search algorithms can be used to locate specific addresses for refund operations.

Below is an example code demonstrating how to use batch processing to avoid gas limitations. We introduce a new parameter, batchSize, in the refundAll() function. This parameter determines the number of addresses processed in each iteration. The loop pauses after processing the specified number of addresses and then continues with the next batch of addresses. This approach prevents excessive data processing in a single iteration.

```
14
15
  contract RefundContract {
       address[] private Addresses;
       mapping (address => uint) public
17
           fundAmount;
       Token private token;
18
19
       constructor(address _token) {
20
           token = Token(_token);
21
22
23
       function addAddress(address_address, uint
24
           _amount) public {
           Addresses.push(_address);
25
           fundAmount[_address] = _amount;
26
       }
27
28
       function refundAll(uint batchSize) public {
29
           uint totalAddresses = Addresses.length;
30
           uint processedAddresses;
31
32
33
           while (processedAddresses <
               totalAddresses) {
               uint end = processedAddresses +
                   batchSize:
               if (end > totalAddresses) {
35
                    end = totalAddresses;
36
37
               for (uint i = processedAddresses;
38
                   i < end; i++) {
39
                        Addresses[i],
                        fundAmount[Addresses[i]])):
40
               processedAddresses = end;
41
           }
42
       }
43
44 }
```

Case 2: The following example [47] illustrates the security issue caused by frequent small transactions on Arbitrum. In a mortgage lending contract, the calculation statement for updating the collateral ratio is as follows:

```
uint change = timeDelta *
    _maxCollateralRatioMantissa /
    _collateralRatioRecoveryDuration;
```

**Security Risks:** By rapidly refreshing or setting the \_collateralRatioRecoveryDuration greater than \_maxCollateralRatioMantissa, it is possible prevent the update of the collateral ratio. Specifically, let's consider a loan pool where the loan token is WBTC and the collateral token is DAI, with each DAI allowing borrowing of only 1/10000 BTC (with a maximum interest rate of \$10,000 per BTC). The \_maxCollateralRatioMantissa is set to 1e14, and the \_collateralRatioRecoveryDuration is set to 1e15. If an attacker makes frequent deposits of 1 wei WBTC within every 10 seconds, the value of (timeDelta \* maxCollateralRatioMantissa) will always be less than \_collateralRatioRecoveryDuration, preventing the update of the collateral ratio. This disrupts the protocol's adaptive pricing mechanism and forces users to borrow at the current interest rate. As the pool's collateral ratio and pool exchange rate will no longer be updated, depositors may experience loss of funds.

#### Risk Avoidance Methods 2:

To address the vulnerability mentioned above, which involves frequent deposits of 1 wei WBTC preventing the update of the collateral ratio, the following measures can be implemented to mitigate the issue:

- 1.Adding a minimum deposit amount restriction: Set a minimum deposit amount to prevent frequent small deposits. By establishing a reasonable minimum deposit amount, any deposits below this threshold will be rejected.
- 2. Adding a deposit time interval restriction: Limit the time interval between deposits to prevent frequent deposits. Set a reasonable time interval, such as allowing only one deposit per hour.
- 3. Implementing a deposit threshold: Introduce a deposit threshold that requires the deposit amount to reach a certain threshold in order to update the collateral ratio.

Here is an example code that checks for the minimum deposit amount, deposit threshold, and cooldown period when a user makes a deposit. It also updates the collateral ratio and the depositor's last deposit time if all conditions are met.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
4 contract CollateralRatioProtection {
      uint256 private constant
          MIN_DEPOSIT_AMOUNT = 1e8; // 1 BTC
      uint256 private constant DEPOSIT_THRESHOLD
          = 1e6; // 0.01 BTC
      uint256 private constant DEPOSIT_COOLDOWN
          = 1 hours:
      mapping(address => uint256) private
9
          lastDepositTime;
      function deposit(uint256 amount) external {
11
          require(amount >= MIN_DEPOSIT_AMOUNT,
               "Deposit amount too small");
          require(amount >= DEPOSIT_THRESHOLD,
               "Deposit amount below threshold");
          require(block.timestamp >=
              lastDepositTime[msg.sender] +
              DEPOSIT_COOLDOWN, "Deposit_
              cooldown period not lelapsed");
          // Update collateral ratio and other
              necessary actions
17
          lastDepositTime[msg.sender] =
19
              block.timestamp;
      }
21 }
```