NFTiX: A Non-Custodial Decentralized Ticket Engine Enabling Secure and Regulated Ticket Exchange Between Untrusted Parties

Yann Gabbud
yann.gabbud@epfl.ch

Distributed Computing Laboratory
and
Secutix SA, an ELCA company

Master Thesis
March 3, 2022

Approved by the Examining Committee:

Prof. Rachid Guerraoui
Academic Supervisor

Mr. Denis Komarov
Company Supervisor

Acknowledgments

I want to express my gratitude to my supervisors, Prof Rachid Guerraoui and Mr Denis Komarov, who gave me the opportunity to work on that project. I want to thank Secutix and the whole TIXnGO team for the fantastic opportunities I had during the internship and the trust they placed in me. Finally, I want to thank Nicolas Plancherel for his guidance and support.

Lausanne, March 03, 2022 Yann Gabbud

Abstract

Players in the ticketing industry are constantly fighting against the black market and fraud to ensure that malicious people do not harm their customers. In recent years, blockchain-based digital tickets have started to become increasingly popular. The idea is to store tickets on the blockchain to guarantee their authenticity and integrity. Two questions now arise. How to ensure that ticket resales are secure and regulated, and how to build a decentralized and non-custodial marketplace.

This thesis seeks to show that it is possible to build the foundations of a decentralized, non-custodial and regulated marketplace based on a user management system, a ticket management system and a ticket exchange built with smart contracts, oracles and indexers. Our system comprises four smart contracts that manage users' registration, enforce access control, apply event and ticket business logic, and allow secure and regulated ticket exchanges between untrusted parties. Our system is also composed of three oracles that complement smart contracts. The oracles securely bring data on the blockchain and execute the business logic that cannot be executed on-chain.

In addition, we present a new approval mechanism for the ERC20 and ERC721 standards that allows a smart contract to operate assets without making an approval transaction beforehand. This mechanism is handy when a task contains several subtasks requiring approval because it allows atomic execution of all subtasks in a single transaction. Moreover, it prevents a well-known attack exploiting a race condition between multiple approvals and transfers due to transaction ordering.

Our contribution is necessary because current solutions are mostly not decentralized and do not fully utilize the potential of blockchain technologies. Moreover, having a regulated marketplace is necessary and will undoubtedly become the norm in the coming years when governments begin to regulate blockchain technologies.

Contents

Chapter 1

Introduction

The ticketing industry is a big market whose capitalization is expected to reach around 60 billion USD by 2026. Every year, people worldwide buy tickets for football matches, concerts or festivals. Unfortunately, sometimes they pay too much for a ticket or buy fake tickets sold by dishonest people.

A ticket is usually a simple PDF with a QR code and some information such as the event's start time or the location of the event. This PDF is generally sent by email to a spectator [1] who prints it or keeps it on his phone. Once at the event, the spectator shows his ticket, which the staff scans. If the ticket is valid, the spectator can enter.

This approach is straightforward and practical for both the spectator and the event organizer. However, it is also very insecure. Indeed nothing prevents someone from buying a ticket and then reselling it to several different people. People who purchase the ticket have no way of knowing that there are multiple instances of the same ticket in circulation. Therefore, although the ticket is valid, only the first person who scans this ticket will enter the event. The others will not be able to enter because the ticket has already been scanned. Note that it is even possible for a dishonest person to forge fake tickets that look deceptively like an original without buying one first.

One solution that event organizers use is to issue name tickets. As the ticket is nominative, it is impossible to forge a fake one. In addition, no one wants to buy a named ticket that is not in their name because they will be denied entry. While this mitigation works, it complicates ticket resale in secondary markets because each time someone wants to resell a ticket, the organizer must delete the old ticket and issue a new one which is inconvenient.

Secutix SA, a famous player in the ticketing industry, started to develop in 2018 a new product called TIXnGO, which aims to prevent fraud. The goal of TIXnGO is to replace traditional paper tickets with digital tickets recorded on a blockchain in a smart contract. Here is an overview of how it works. A user buys a ticket from an organizer. Instead of receiving the ticket by mail,

[1] You can find more information about the ticketing terminology in the background section.

the user downloads the TIXnGO mobile application and registers by giving and verifying his email address. Once the user is registered, the digital ticket is created and injected into a smart contract. Finally, TIXnGO notifies the user that his ticket is available on the application. Once the day of the event has arrived, the user presents his smartphone to the access control, which lets him enter.

This approach works very well to fight against fraud because only an organizer registered on TIXnGO can inject tickets on the blockchain. A malicious user cannot forge fake tickets and put them into circulation because the only way for him to do so is to register on TIXnGO and thus reveal his identity.

However, the approach is insufficient because it is only partially decentralized and does not fully utilize blockchain technology. Indeed, it only uses the blockchain to store tickets. It certainly brings security and transparency, but a blockchain can be more than a secure database. Moreover, their approach is custodial, meaning users do not own their tickets and cannot use their crypto wallets to store their tickets. Finally, this approach only partially addresses the problem of the black market. Indeed, nothing prevents a dishonest user from amassing numerous tickets and reselling them at high prices. TIXnGO monitors all ticket movements to fight the black market by detecting suspicious behaviours. It is easy to do because the blockchain logs all transactions. Therefore, it is possible to know who the original purchaser of a ticket is, who is the current owner of a ticket and who the previous owners are. TIXnGO analyzes this data and triggers alerts when illegal behaviours are detected. If necessary, it takes measures.

Ideally, the ticketing industry needs a user management system, a ticket management system, and a ticket exchange that are fully decentralized, transparent, secure, and GDPR compliant. In addition, the exchange must make it possible to easily resell tickets while allowing market regulation through transfer and resale rules [2] to fight the black market.

However, building such a system is not easy. The first challenge comes from the immutability of the logic of smart contracts. Once a smart contract is deployed on the blockchain, it is no longer possible to modify it. It is a massive constraint because all the logic, including the transfer and resale rules, must be planned and cannot change later. It is necessary to deploy a new smart contract to update the logic. Deploying a smart contract can be very expensive. On the Ethereum blockchain, the cost of deploying a smart contract can go up to several thousand CHF in the event of heavy network congestion. Unfortunately, it is impossible to plan everything because the rules often change over time. What was relevant in the past is not necessarily relevant now or in the future. Here is an example. In 2021 the English government imposed on event organizers to restrict the transfer and resale of tickets to only English citizens to limit the number of foreigners entering the country and, therefore, fight against the spread of Covid19. This kind of event is typically impossible to anticipate. Therefore, to manage this new case, deploying a new smart contract that supports the new constraints and pays again is necessary. It is not practical in an actual situation. Therefore, we need an easily scalable system.

[2] A resale rule is for example, the resale price of a ticket cannot exceed more than 20% of the initial price.

The second challenge is the size of a smart contract. A smart contract cannot exceed a specific size limit; otherwise, it is impossible to deploy it. Therefore, it limits the complexity of a smart contract's logic. The problem is that organizers have a lot of different logic and rules depending on the event's location or type. In addition, these rules may vary from one organizer to another. For instance, the organizers of the Wimbledon championships have defined more than a hundred different transfer and resale rules. It is impossible to implement all of them in a single contract. Nonetheless, it is possible to circumvent this limitation by having a main contract calling secondary contract methods. However, although this trick allows for more richness in logic, it does not allow for arbitrary high complexity, and the operating cost of such an approach is very high.

The third challenge comes from the fact that the blockchain, and therefore the smart contracts, do not have access to external data. Unfortunately, calling a database from a smart contract is not possible. One solution would be to store all data directly on the blockchain. However, it would be very costly. Moreover, according to the GDPR law, it is forbidden to store data concerning a user on the blockchain. The organizers do not want either that some of their data are public. Therefore, smart contracts cannot implement all the logic due to the law and the lack of available data.

Finally, there are three other challenges. First, the system must support a large number of simultaneous operations. In one year, the system can manage up to several million tickets. It represents several thousand operations per day to be processed. Second, operation confirmation time must be short. For example, the system must immediately acknowledge scanned tickets so that it is not possible to enter the event twice with the same ticket. Finally, it must be possible to make complex and heavy queries about the system's state. For example, get all scanned tickets and sort them. Current blockchains are known not to be very good at these exercises at the moment. However, thanks to layer 2, such as zk rollups, significant progress has been made, which bring scalability without compromising security.

Since it is impossible to implement and, by extension, execute arbitrary complex logic on the blockchain, we suggest executing some of the logic outside the blockchain and verifying on-chain using proof that the component responsible for this has done its due diligence. To do this, we propose the following approach. A database stores the data necessary for the logic execution and the transfer and resale rules. For each transaction, an oracle checks that the rules are respected. The oracle then issues proof that the blockchain is authorized to execute the transaction. When the user creates the transaction, he attaches the proof issued by the oracle. During the execution of the transaction, a smart contract verifies the proof. If the proof is not valid or the oracle has not issued the proof, the transaction is refused. Otherwise, the transaction is executed.

With this approach, we can have smart contracts much simpler, supporting any transfer or resale rules and less greedy in terms of gas. Indeed, the smart contract only verifies that the proof is correct and does not need to execute a complex logic to approve or refuse the transaction.

Here is an example illustrating how our approach works. Suppose a spectator wants to resell a ticket. He bought it for 100 CHF and wants to resell it for 115 CHF. Suppose that the event organizer also imposes that the resale price is not higher than 20% of the initial price. As the resale price is less than 120 CHF, the resale is authorized. Here is the procedure. First, the spectator sends his request to the oracle, which checks that the rules are respected. The request contains all the information that will be used to create the resale transaction on the blockchain, such as the buyer's identity and the resale price. The Oracle uses the information in the request to create a transaction approval message, sign it, and return it to the user. The user creates the transaction with the information he has previously provided to the oracle and the approval message signed by the oracle. It sends the transaction to the blockchain, which verifies it. A smart contract that knows the oracle's public key ensures the verification. Finally, the transaction is executed.

Implementation and testing show that it is possible to build such a system. The evaluation shows that it could be possible to deploy it in production. First, our approach is scalable. Indeed, an oracle running on a simple laptop easily supports 1,500 requests per second. The main bottleneck is still the blockchain. Second, verification on the blockchain of the oracle's message leads to a relatively small increase in transaction costs, about 10%. Note that if we were to run all the logic on the blockchain, the cost increase would be much higher. A simulation of the system on Polygon shows that the cost of performing a resale transaction is less than 1 cent, which is perfectly bearable for the ticketing industry or the users.

Therefore, our approach makes it possible to build a regulated marketplace that supports any regulation rules. In addition, as the execution is carried out or verified on the blockchain, we ensure a very high level of security and decentralization. The only downside comes from the oracle, which is unfortunately not decentralized. However, decentralized oracles, such as Chainlink, exist and can achieve a fully decentralized system.

In summary, our main contribution is to lay the foundations for building a secure, regulated, non-custodial and decentralized marketplace based on a user management system, a ticket management system and a ticket exchange. Additionally, we present a new approval mechanism for the ERC20 and ERC721 standards that allows atomic execution of complex tasks and prevent attacks exploiting a race condition on approve and transferFrom methods. We present a few other mechanisms that can be interesting for the ticketing industry, such as time-based blockchain logic or a mix of custodial and non-custodial design, allowing users to control their assets while ensuring regulation and a high level of security. Finally, we want to clarify that although this approach uses the ticketing industry as a scenario, it can be perfectly applied to other sectors with similar needs.

Chapter 2

Background

In this section, we introduce the background needed to understand this thesis. We first present the ticketing industry and the vocabulary it uses. Then we introduce the technologies and dependencies to understand the design and implementation.

2.1 Ticketing terminology

Digital Ticket: A digital ticket is nothing more than a digital representation of a paper ticket. However, it has many advantages. First, it is more transportable and much more easily transferable or resalable than its paper counterpart. It is also more difficult to create fake digital tickets because they are generally cryptographically signed and governed by a system with an access control preventing anyone from putting them into circulation without approval. In our case, we use the ERC721 standard to represent digital tickets. This standard works well for digital tickets because it allows us to create unique transferable assets and track who owns them.

Ticket wallet: A ticket wallet is an application that stores digital tickets and displays them when requested by the user. In general, a wallet is a mobile application like the Apple wallet or a wallet dedicated to tickets like the TIXnGO wallet.

Event: An event is a generic term that encompasses sporting events, festivals, concerts, or any other event that you need a ticket to attend.

Organizer: An event organizer, or organizer for short, is a person or entity that creates, manages, and promotes events and sells tickets for them.

Spectator: A spectator is someone who purchases tickets and attends events. A spectator may also transfer or resell tickets he has purchased from an organizer.

User: Sometimes, this term is used to encompass both organizers and spectators who are registered in the system.

Ticket Transfer: A spectator who owns a ticket can transfer it to someone else. This person becomes the new owner of the ticket. This change of ownership is called a ticket transfer.

Ticket Resale: A spectator who owns a ticket can resell it to someone else. It is similar to a ticket transfer except that there is also a money transfer in addition to the ownership transfer. We say the current owner of the ticket is the seller and the person buying the ticket is the buyer. Therefore the seller transfers the ticket, and the buyer transfers the money.

Ticket Swap: A spectator who owns a ticket can swap it against another one. It is similar to a ticket resale, except that two tickets are exchanged instead of a ticket against money. We call the two parties of a swap A and B, or Alice and Bob.

Transfer, resale and swap rules: Operations on tickets are subject to rules defined by the organizer. These rules define a policy that mainly aims to fight against the black market and fraud. An example of a transfer rule might be that it is prohibited to transfer a ticket to a fan banned for hooliganism. An example of a resale rule might be that the resale price of a ticket cannot exceed more than 20% of the original price.

Ticketing system: A ticketing system, like Secutix, is a ticket management tool. An event organizer uses a ticketing system to create and manage events, create, manage and sell tickets for an event, and provide support to spectators who purchase tickets for its events.

Ticket Distribution System: A ticket distribution system, such as TIXnGO, is a tool for managing the distribution of digital tickets. A ticket distribution system does not replace a ticketing system but is complementary. Often it also helps to fight against fraud and the black market. In general, there are three main components. The first component is a management interface used by the organizer to distribute tickets purchased by spectators and track who the current owners are. The second component is a ticket wallet used by the spectator to store, display, transfer and resell tickets. The third component is a ticket engine that executes the requests of the organizers and the spectators. It is important to note that only a registered organizer can use the management interface. Therefore, only a genuine organizer can distribute tickets such that no fake tickets are in circulation.

Marketplace: A marketplace is a platform that allows the secure sale of assets between parties that do not trust each other. In our case, the assets are tickets. The marketplace displays tickets that are on resale with their resale price. A seller can put a ticket up for resale and withdraw it later if he changes his mind. A buyer can purchase a ticket on resale.

2.2 Ethereum blockchain

Ethereum is an open-source blockchain [1] that runs smart contracts. It aims to be a global internet computer for building decentralized applications [2]. It is currently the main blockchain used for Defi [3], NFTs and Dapps.

The system we design is based on the Ethereum blockchain. However, note that our implementation is deployable on any EVM-compatible blockchain, such as Avalanche, Polygon or Binance smart chain. Note that it is perfectly possible to implement the design to deploy the system on blockchains that are not EVM-compatible but support smart contracts such as Solana or Cardano.

2.3 Ethereum wallet

An Ethereum wallet is an application that lets a user interact with the blockchain. With his wallet, the user can manage his account and assets. For instance, he can get his ETH balance. He can make a transaction to transfer ETH to someone else. He can also connect and access decentralized applications. Wallet stores public and private keys. The public key is used to identify the account and, by extension, its owner. The private key is used to sign transactions and ensure that only the wallet owner can interact with his account. More information here [4].

2.4 Custodial vs non-custodial

As explained in the previous section, the wallet contains a private key to sign transactions. The difference between a custodial approach and a non-custodial approach is minimal but of great importance. In a custodial approach, the platform holds the user's private key, while in a non-custodial approach, the user holds his private key. What must be understood is that when the user owns his private key, he own his funds!

[1] https://en.wikipedia.org/wiki/Blockchain
[2] https://en.wikipedia.org/wiki/Decentralized_application
[3] https://en.wikipedia.org/wiki/Decentralized_finance
[4] https://ethereum.org/en/wallets/

2.5 Transaction

A transaction is an operation initiated by an external account [5] that changes the state of the blockchain. An example of a transaction is a transfer of ETH from one person to another. Since the transaction changes the state of the blockchain, the account that initiates the transaction must pay a transaction fee to the blockchain. In the example, we see that the state of the blockchain is changed because the transferred amount decreases the sender's balance, and it increases the receiver's balance. More information about transactions here [6].

2.6 Call

A call is an operation on the blockchain that does not change the state of the blockchain. Both external and contract accounts can make calls. For example, one can make a call to get his ETH balance. Since a call is read-only, the call initiator does not have to pay fees. More information about calls here [7]

2.7 Smart contract

A smart contract, or contract for short, is a program that runs on the blockchain. As the blockchain is Turing complete, a smart contract can implement any arbitrary logic. However, there are two limitations to keep in mind. The first one is that a smart contract cannot call an external service and only has access to data stored on the blockchain or sent with a transaction. The second is that a smart contract cannot implement arbitrary complex functions. Indeed if a function exceeds a certain complexity, the function will be reverted by the EVM [8] because it exceeds the gas limit [9].

2.8 ERC20

ERC20 [10] is the Ethereum standard for creating fungible tokens. It defines a set of rules that any token must implement to be compatible with the standard. The main feature of such a token is fungibility. It means that the tokens are indistinguishable from each other. For example, fiat

[5]https://ethereum.org/en/developers/docs/accounts/
[6]https://ethereum.org/en/developers/docs/transactions/
[7]https://ethereum.stackexchange.com/a/770
[8]https://ethereum.org/en/developers/docs/evm/
[9]https://ethereum.org/en/developers/docs/gas/what-is-gas -limit
[10]https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

currencies, such as CHF, EUR or USD, are fungible. Any CHF 100 note can be exchanged for another CHF 100 note. Stock auctions are also fungible. Thus, this standard aims to make it possible to represent any fungible asset on the Ethereum blockchain.

2.9 ERC721

ERC721 [11] is a standard for non-fungible tokens. Like the ERC20 standard, ERC721 is a popular standard. However, it has a different purpose. It aims to represent any non-fungible asset on the Ethereum blockchain. A non-fungible asset can be, for example, an event ticket. Indeed, each ticket has a unique identifier (UID) that distinguishes it from any other ticket for the same event. The Venus of Milo is another example of a non-fungible asset. There is only one Venus that Paros sculpted.

2.10 Smart contract event

An event is a piece of information emitted and logged by the blockchain when a transaction is mined. For example, when someone transfers ERC20 tokens, a *Transfer* event is emitted. This event includes the following information: the sender's address, the receiver's address and the amount sent. Events emitted by a smart contract can be used to build indexes to perform complex data queries.

2.11 Blockchain oracle

As mentioned earlier, smart contracts do not have access to data stored outside of the blockchain. Therefore, blockchain needs a way to bring data securely onto the chain. It is the role of an oracle. It guarantees the authenticity and integrity of this data so that the blockchain can access off-chain data securely without compromising its security.

2.12 Keccak256

Keccack256 is one of the most widely used hash functions in the Ethereum ecosystem. This hash function is used in the EVM and in Ethash, the PoW algorithm of Ethereum. Note that this hash function is named keccak256 and not SHA3 [12] even though they are nearly identical.

[11]https://ethereum.org/en/developers/docs/standards/tokens/erc-721/
[12]https://en.wikipedia.org/wiki/SHA-3

The reason is that keccak256 is an old version of SHA3. SHA3 was slightly modified in August 2015, right after the launch of Ethereum in July 2015. The oracles use keccak256 to calculate the approval message from user inputs in our system.

2.13 ECDSA

ECDSA, Elliptic Curve Digital Signature Algorithm, is used to sign messages digitally. This algorithm is very suitable for blockchains because, compared to its counterparts such as RSA, it uses shorter keys, has faster signing and encryption operations, and offers the same security level. Oracles use ECDSA to sign the messages they computed with keccak256.

2.14 Graph protocol

The Graph [13] is a protocol used to index blockchain and smart contracts. It is handy because it allows to build and maintain off-chain indexes tracking the state of the blockchain or smart contracts. Building an off-chain index is very useful because it simplifies access to information and reduces the complexity of smart contracts. There is no need to create and maintain complex data structures in the smart contract in order to be able to query arbitrary data. It also allows querying data in bulk, which is impossible with a blockchain call due to the gas limit. Note that a traditional database such as Postgres stores indexes.

[13]https://thegraph.com/docs/en/about/introduction/

Chapter 3

Related Work

Many projects seek to create blockchain-based digital tickets and marketplace such as B.A.M, Blocktix. EventCHI, FanDragon, PassageX, TicketHash, TIXnGO or GET protocol. For most of them, it is not easy to find out how they work because they do not have documentation open to the public, and the product description is aimed primarily at the marketing guys. We can say that they all offer a simple form of regulation, such as limiting the maximum resale price to fight against the black market. They all have a custodial model, i.e. they handle tickets for users. They do not appear decentralized and primarily use blockchain as ticket storage.

We detail two systems similar to ours that seem to be the most promising and have available documentation: TIXnGO and GET Protocol. We begin by giving an overview of these systems. Then we explain the key differences in our approach and how we overcome some of their weaknesses.

Let us start by giving an overview of the two systems. They are both ticket distribution systems. Secutix SA develops TIXnGO, and GUTS ticketing develops GET Protocol. Their approach is to use the blockchain to securely and transparently store tickets and their owners. Now let us take a look at how they work. To do this, we take the example of Fifa, which organized the Arab Cup last year, and we detail the flow from the purchase of tickets until the spectators enter the stadium. First, Fifa sells tickets to spectators through its website. Once the tickets are sold, the distribution system sends a list with the owners and ticket details. The distribution system saves everything in a database and stores ticket ID and spectator ID pairs in a smart contract. Once done, it informs the spectators that their tickets are available on the mobile app. From then on, spectators can transfer their tickets to other spectators registered on the system. This feature is handy because usually, someone buys tickets and then gives them to his friends or family. On match day, spectators gain access to the stadium by scanning their tickets at the entrance.

Let us now take a closer look at TIXnGO. The approach fulfils its role well, and the partners are pleased with the product. However, there are many things that we believe can be improved. The first point concerns the blockchain. TIXnGO uses a private blockchain hosted on Amazon servers. Having a private blockchain is very convenient because there are no transaction fees to pay. It is possible to quickly correct manipulation errors and a bug in smart contracts. However, a private blockchain does not make much sense because no one except TIXnGO has access to it. Organizers and spectators use an API but never communicate directly with the blockchain. Therefore, most of the exciting properties of the blockchain, such as transparency or direct management of assets, are lost. In order to fully benefit from the advantage of the blockchain, we based our approach on a public blockchain.

The second point concerns business logic and general design. Almost all the business logic of TIXnGO is managed off-chain by a backend. The blockchain only manages the ownership of the ticket. This approach is convenient because it is easier to implement off-chain business logic. For instance, smart contracts cannot be arbitrarily complex, which is a significant limitation in some situations. However, we believe that the blockchain should take care as much as possible of the business logic in order to take advantage of the blockchain and not just use it as a database that is certainly very secure but also very expensive to operate. It would also make it possible to have a much more decentralized system and avoid a single point of failure. Our approach aims at executing as much of the business logic as possible on-chain. The business logic that cannot be executed on-chain is executed off-chain and verified on-chain. To do this, we use oracles, smart contracts.

The third point concerns standards. TIXnGO uses homemade smart contracts that standards could replace. For example, using the ERC721 standard to represent tickets and keep track of their owner would be interesting. The problem for TIXnGO is that using this standard requires many system adjustments because it does not use Ethereum addresses as identifiers for users but database IDs. Our approach tends to use as much as possible the current standards. It allows us to benefit from the work carried out by the blockchain community and maximize our system's interoperability with other projects.

The fourth point, which we think is the most important, concerns custody. TIXnGO uses a custodial approach. In general, this approach is preferred by companies because it is simpler to implement and economically advantageous, especially for an exchange. It allows TIXnGO to control the entire process and, therefore, quickly intervene in the event of a problem. However, as we mentioned previously, this approach implies that the user is not the owner of his assets because he delegates their management to TIXnGO. However, the original purpose of a blockchain is to avoid this by giving control back to the user. Therefore, we chose to have a non-custodial design. Although this approach is more challenging to build, we show that it is possible to design a system that achieves the same objectives as a custodial system and gives control back to users! We design an on-chain user management system and on-chain role-based access control to achieve this objective.

Finally, although TIXnGO allows the resale of tickets, it does not have a marketplace yet. We propose to build the foundation of a fully decentralized marketplace allowing secure and regulated resales of tickets.

Now let us take a closer look at the GET protocol and compare it to TIXnGO. As the systems are very similar, we do not go into details not to repeat what was said above. About the first point, unlike TIXnGO, it is deployed on Polygon, a public blockchain. It is an excellent thing because it gives visibility to the system to the users. About the second point, we believe it makes better use of the blockchain. In addition to tickets, it stores events on the blockchain. It is a good idea because it makes it easier to build an index and filter tickets efficiently by events. However, we believe that, like TIXnGO, the system does not take full advantage of what the blockchain has to offer. About the third point, it should be noted that it uses the ERC721 standard to represent tickets on the blockchain. Due to the popularity of this standard, this is a good design choice as it allows the system to interoperate with other systems easily. On the fourth point, it also uses a custodial approach. All interactions with the system are done through an API. Finally, like TIXnGO, it does not yet have a decentralized marketplace.

To conclude, we must briefly mention Taurus, a fast-growing Geneva startup. Taurus offers financial market infrastructure services and products such as a custody solution used by Swiss banks. This solution allows banks to offer their customers the possibility of buying crypto assets. On top of that, they are building a regulated market, named TDX, to enable the trading and exchange of tokenized securities which is an essential and missing piece in the financial ecosystem and the first in the world, as far as we know. The marketplace complies with Swiss securities law and Distributed Ledger Technology (DLT) law.

Although the business is different, it is interesting that a Swiss actor seeks to regulate decentralized assets as we seek to do. Unfortunately, there is no documentation, and the API is not yet available. Therefore, it is difficult to understand how it works under the hood, except that it is custodial.

Chapter 4

System overview

In this chapter, we first enumerate the system's goals. Then, we describe the system's components. We made several iterations before coming up with the final version. Therefore, we describe the initial design for each component and the improvements that we made.

4.1 System goals

NFTiX is a ticket engine that seeks to facilitate ticket exchange between untrusted parties and provide the following advantages in terms of decentralization, scalability and security.

Non-custodial: The users own and control their assets. NFTiX provides the service but cannot manipulate user assets.

Decentralized: There is no single point of failure in NFTiX such as a trusted third party.

Regulated: NFTiX has a mechanism that allows controlling the resale process so that the black market is limited.

High Throughput: NFTiX supports thousands of daily operations such as tickets issuance, transfers, or resales.

Low Latency: NFTiX offers low latency transaction confirmation.

Low Costs: The operating cost of NFTiX is low for developers, event organizers and spectators.

Integrity, authenticity and unforgeability: NFTiX ensures a high level of security so that the integrity, authenticity and unforgeability of tickets is guaranteed and no fraud is possible.

4.2 Architecture overview

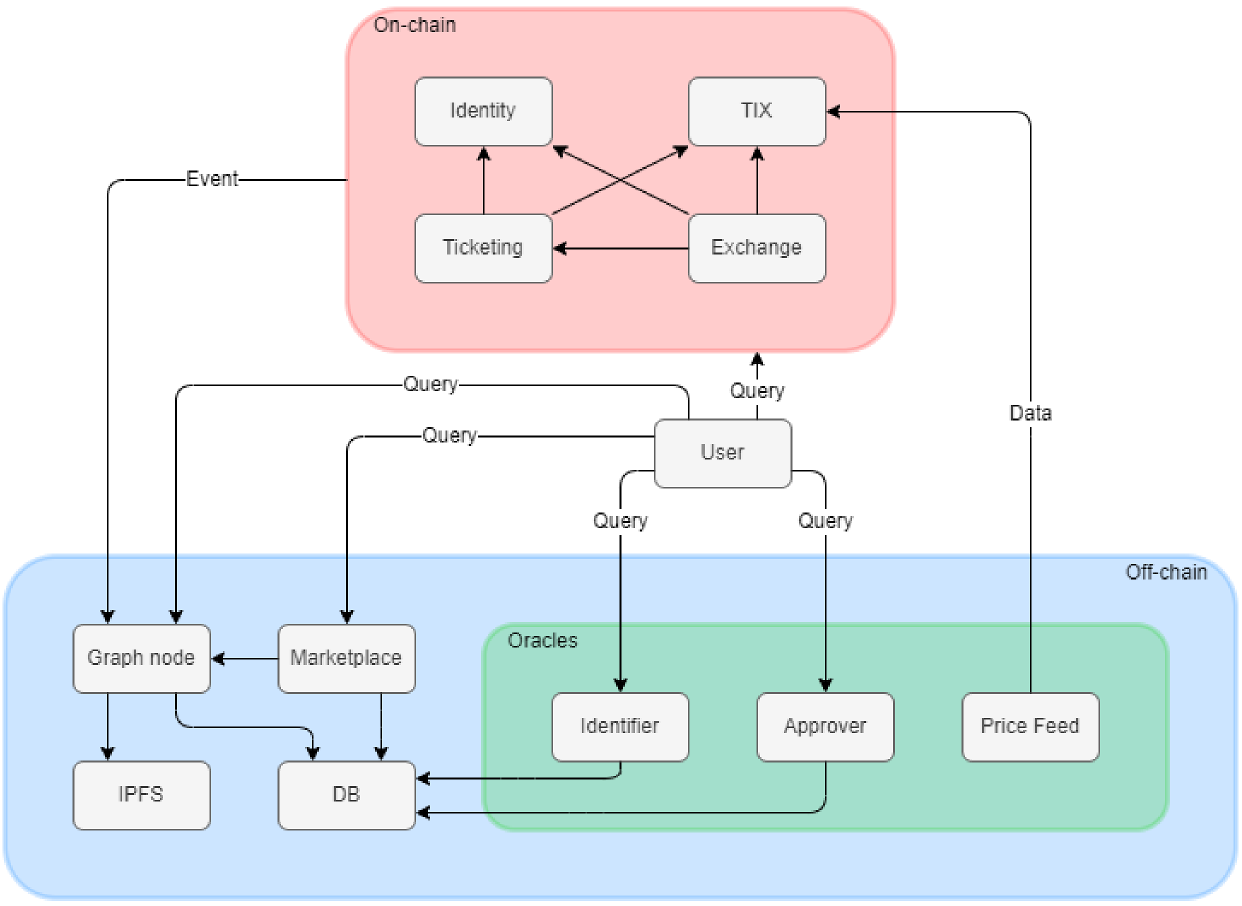


Figure 4.1: Architecture overview

This section aims to overview the main components and their interoperability. As you can see on the figure 4.1, the system is split in two parts: on-chain and off-chain. The on-chain part contains four smart contracts that enforce the business logic. The Identity contract tracks the registered users and enforces the access control policy. The Ticketing contract manages events and tickets. The Exchange contract allows spectators to resell and swap tickets. The TIX contract is an ERC20 utility token that powers the economics of NFTiX. The TIX is used to pay the platform's fees and purchase tickets.

The off-chain part contains the oracles, which are crucial because they securely execute the business logic that cannot be executed on-chain. The Identifier verifies the new user's identity and allows them to register on the Identity contract. The Approver verifies that the organizers approve the operations on tickets. The price Feed is responsible for bringing on-chain the price of the TIX against the ETH and the USD.

Finally, the graph node is responsible for indexing the smart contract such that we can easily query data. The marketplace is responsible for listing the resale and swap offers. The following sections give a more detailed description of each component.

4.3 Oracles

Oracles play a crucial role in the system. There are two types of oracles in our system. The first two, the Identifier and the Approver, execute the business logic that cannot be executed on-chain and issue approval messages verified on the blockchain. The third, the Price Feed, takes care of sending data on-chain, in this case, the price of the TIX against the USD and the ETH.

Note that all the oracles have a pair of private/public keys. Transactions and approval messages are signed and verified with these keys.

A message issued by an oracle is proof that the oracle has done its due diligence. For example, in a resale, the Approver certifies that it has checked that the operation complies with the resale rules. Let us look at how such proof is constructed. The creation takes place in three steps. First, the oracle checks that the operation is legal. Then, the oracle hashes the arguments supplied by the user with the help of the hash function Keccak256. Once the oracle has computed the hash of the arguments, it creates a signature with its private key and the hash using ECDSA. This signature is the proof returned to the user.

4.3.1 Identifier

The Identifier verifies the identity of the organizers and the spectators who register in NFTiX, stores user information on the database, and issues identification proofs allowing users to register on the Identity contract.

Currently, the oracle only verifies the Ethereum address that the user provides. The oracle sends a challenge that the user must sign and return to the oracle. The verification is successful if the oracle manages to verify that the private key that signed the message corresponds to the address provided by the user. As the name suggests, the oracle should also verify the user's identity to comply with the KYC rule. This verification mechanism is not implemented because outside of the scope of the project. Therefore, we assume that the oracle performs an identity verification similar to that carried out during the onboarding of Coinbase, Binance or Swissborg, i.e. the user sends photos of his identity card and face, which are then verified thanks to machine learning.

Once the verification is successful, the oracle saves the user's information and issues an identification proof built from the user's Ethereum address and group. The group allows to distinguish spectators from organizers and build the role-based access control. We will come back to this later.

4.3.2 Approver

The Approver is responsible for verifying that the operations on tickets initiated by spectators comply with the rules imposed by the organizers and for issuing proofs of approval allowing spectators to carry out resale and swap transactions. We explain later how resale and swap transactions work.

For a resale transaction, The proof is constructed with the address of the ticket owner, the token ID (remember that we use non-fungible tokens to represent tickets on the blockchain), the resale price and the address of an optional buyer chosen by the seller. The proof is constructed with the zero address if the optional buyer is not specified. For a swap transaction, the proof is constructed with the address of spectator A, the address of spectator B, the ID of token A and the ID of token B.

The approval procedure of a transaction works like this. First, the spectator sends a transaction approval request to the Approver, providing it with the data used to construct the transaction. The oracle verifies that the transaction respects the rules and issues transaction approval proof. The spectator then uses the proof and the data he transmitted to the oracle to construct a transaction.

4.3.3 Price feed

The Price Feed regularly sends the price of TIX against USD and ETH to the TIX smart contract. As a reminder, smart contracts cannot communicate with the outside world, and therefore the TIX smart contract has no way of knowing the price of the TIX if it does not receive it.

This oracle works differently from the other two oracles. It communicates directly with the blockchain and does not need to issue proof. Indeed, as transactions are signed, the smart contract only needs to verify that the initiator of the transaction is the Price Feed to ensure that the price update is legitimate. Note that the TIX smart contract stores the address of the Price Feed in order to verify that the transaction comes from it and not from a malicious user.

4.4 Ethereum blockchain

The Ethereum blockchain is the cornerstone of the system. It is responsible for operating the smart contacts that enforce the business logic.

This component must be robust and battle-tested to ensure that the execution environment is safe. We chose Ethereum because it is the most mature blockchain at the moment. It has been extensively tested, hosts many projects, and has a complete development and test environment.

However, it has a significant disadvantage, the high cost of transaction fees induced by the congestion of its network. Unfortunately, it suffers from its popularity. However, many scalability solutions are under development, and things should improve. In the meantime, it is possible to deploy the system on Polygon, which is EMV-compatible and offers a good security level.

4.5 Smart contracts

This section describes the core of NFTiX, which is responsible for enforcing the business logic. The core is made up of four smart contracts: the Identity contract, the TIX contract, the Ticketing contract and the Exchange contract. For each contract, we describe first the initial design, then the improvements made to correct the initial design errors. There are several flow diagrams with highlighted flows. The flows highlighted in red are off-chain operations, in blue are transactions sent to the blockchain, and in green are on-chain operations.

4.5.1 Identity smart contract

The Identity contract takes care of several things. It allows the organizers and spectators to register and unregister from NFTiX. It allows NFTiX to revoke an organizer or a spectator who misbehaves. It stores the addresses of registered organizers and spectators. It stores the Identifier's public key to verify the identification proofs issued by the oracle when an organizer or spectator registers. The figure 4.2 illustrates the flow of user registration.

This contract also defines and applies the role-based access control policy. There are four roles: ORGANIZER, SPECTATOR, UNREGISTERED and REVOKED. An organizer who registers gets the ORGANIZER role, and a spectator gets the SPECTATOR role. When an organizer or spectator unregisters, their address is marked as UNREGISTERED. If NFTiX bans a user from the system for inappropriate behaviour, his address is marked as REVOKED.

With these roles, we can distinguish the organizers from the spectators. It allows us to give high privileges to the organizers and limited privileges to the spectators. In addition, listing registered spectators allows us to limit exchanges to registered spectators only. Note that we sometimes use the name group instead of role but it means the same thing in this context.

4.5.2 TIX smart contract

This contract manages the utility token of NFTiX, the TIX. This contract is ERC20 compliant. The TIX is used to pay platform usage fees and purchase tickets on resale. Note that the platform generates revenue through the activity of its users. For example, when an organizer creates an event, he must pay a registration fee. When he mints a token, he must pay a minting fee. When

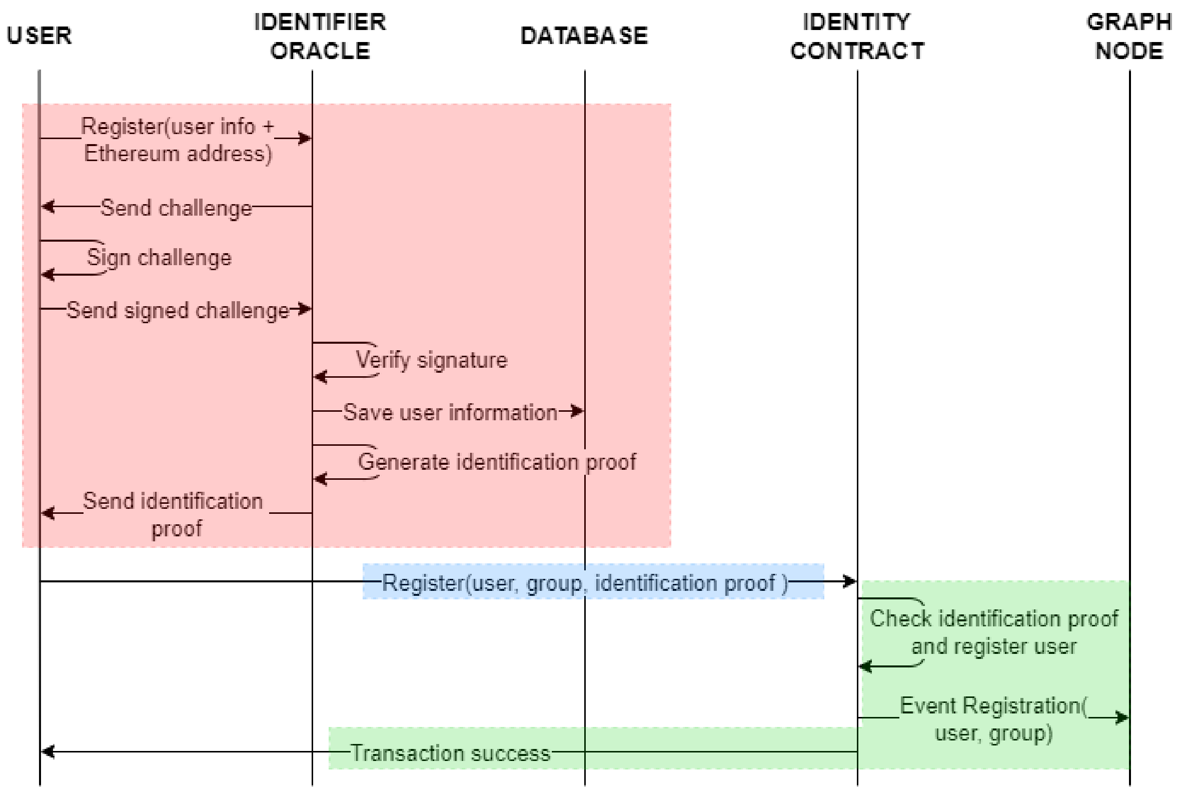

USER
IDENTIFIER ORACLE
DATABASE
IDENTITY CONTRACT
GRAPH NODE
Register(user info + Ethereum address)
Send challenge
Sign challenge
Send signed challenge
Verify signature
Save user information
Generate identification proof
Send identification proof
Register(user, group, identification proof)
Check identification proof and register user
Event Registration(user, group)
Transaction success

Figure 4.2: User registration

a spectator purchases a ticket on resale, a percentage of the resale price is taken and shared between the event organizer and NFTiX.

The contract stores the price of TIX against dollars and ETH. The price feed oracle regularly updates these prices. The smart contract stores the oracle's public key so that only it can check that the prices update are genuine

Users can request the current price of TIX against dollars or ETH, ask how many TIXs an amount in USD or ETH is worth, buy TIXs with ETH, or sell TIXs and receive ETH in return. The contract mints tokens when a user buys TIXs and burns tokens when a user sells TIXs. Users can also transfer TIXs to someone else. Finally, the admin of NFTiX can request the ETH balance of the contract and remove or add ETH to the smart contract.

It is important to note that the token has no actual use because we could use ETH instead of TIX. However, it does show to Secutix SA how a utility token can be integrated into the system and used by users. In the future work section, we give a few ideas of the potential uses of the token.

Finally, note that we slightly improved the ERC20 standard, which helps reduce the number of transactions needed when a transfer of tokens is part of a task containing many subtasks. It also prevents a well-known attack leveraging a race condition between multiple approvals and transfers due to transaction ordering [1]. This improvement consists of creating an off-chain proof allowing a smart contract to transfer tokens on behalf of a user without performing an approval transaction first. We call this an externally approved transfer. This type of approval is helpful to build the exchange. We will come back to its use later.

The proof, which we call a TIX transfer approval proof, is constructed the same way as the oracles' proof. The user creates the proof with the address of the approved contract, his address, the recipient address and the transferred amount. The proof is then put as an argument in a transaction and verified by the TIX contract. Note that this approval mechanism must be used with caution. We analyze what could go wrong if the mechanism is misused in the security chapter. In the future work section, we describe an anti-replay mechanism that we do not use, but which is useful in some situations.

4.5.3 Ticketing smart contract

This contract manages events and tickets. As NFTs represent the tickets, the contract complies with the ERC721 standard.

Let us start by detailing the events. To represent the event, we use a data structure containing the following fields: an event ID, the address of the organizer, the name of the event, the place, the opening date and time, the closing date and time and finally, the state of the event.

Three of these fields serve practical purposes for business logic: the event ID, the opening date and time, and the state, while the others could be stored off-chain to reduce gas consumption. The event ID is used to distinguish events from each other. This identifier is unique. The opening date and time correspond to the opening of the event's doors. This information is used to prevent transfers and resales of tickets after the event's opening. This measure is necessary to prevent a malicious person from reselling a ticket that has already been scanned.

There are three possible states for an event. The first state is PENDING. This state means that the event has been created but is not yet open. Therefore, an organizer can create tickets for this event. However, it is impossible to transfer, resell or swap them yet. The second state is OPEN. This state means that the event is activated. Therefore, spectators can transfer, resell or swap their tickets. The last state is CANCELED. We use this state to mark an event as cancelled. Transfers, resales and swaps are frozen with this state, as with the PENDING state. Although these two states have the same effects, it is practical to distinguish them to filter events and tickets.

Now let us take a look at tickets. A ticket comprises four fields: a token ID, an event ID, a ticket ID and a state. The token ID is the unique identifier that distinguishes NFTs from each other. The event ID corresponds to the event the ticket belongs to. The ticket ID is a unique identifier

[1]https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA$_{j}p - RLM/edit$

that allows the organizer to distinguish his tickets. Note that two organizers can share the same ticket ID. Therefore, to uniquely identify a ticket, one must either know its token ID or its ticket ID and its event ID. Finally, like an event, a ticket has a state. The first state is VALID. It means that the ticket can be transferred, resold, swapped or used to enter the event. The second state is INVALID, which means that the organizer has invalidated the ticket. There can be several reasons for this. It is a test ticket. Its bearer tried to defraud. Or, the ticket was cancelled for another reason. The last state is SCANNED. It means that the ticket is valid and the bearer has entered the event. This state is practical because sometimes it is necessary to scan a ticket again, such as to access the VIP area.

Let us look at how organizers and spectators can interact with the smart contract. First, note that some smart contract methods, such as event creation, can only be performed by an organizer. If a spectator attempts to execute one of these methods, the transaction will be reverted. To enforce this restriction, the smart contract queries the Identity smart contract to find out if the initiator of the transaction is an organizer. Secondly, note that an organizer can only operate on his events. For instance, it cannot mint a ticket for the event of another organizer. If he does, the transaction is reverted.

An organizer can register a new event. He must provide all the fields listed above and pay the registration fee to do this. This fee, paid in TIX, is automatically taken from the organizer's balance. Therefore, the organizer must first have sufficient TIX and approve [2] the smart contract to transfer the registration fee amount on its behalf. If either condition is not satisfied, the transaction is reverted. Note that if the opening date and time is after the closing date and time, the transaction also fails. If the organizer makes a mistake while registering an event, he can update its fields later. The figure 4.3 illustrates the flow of event registration.

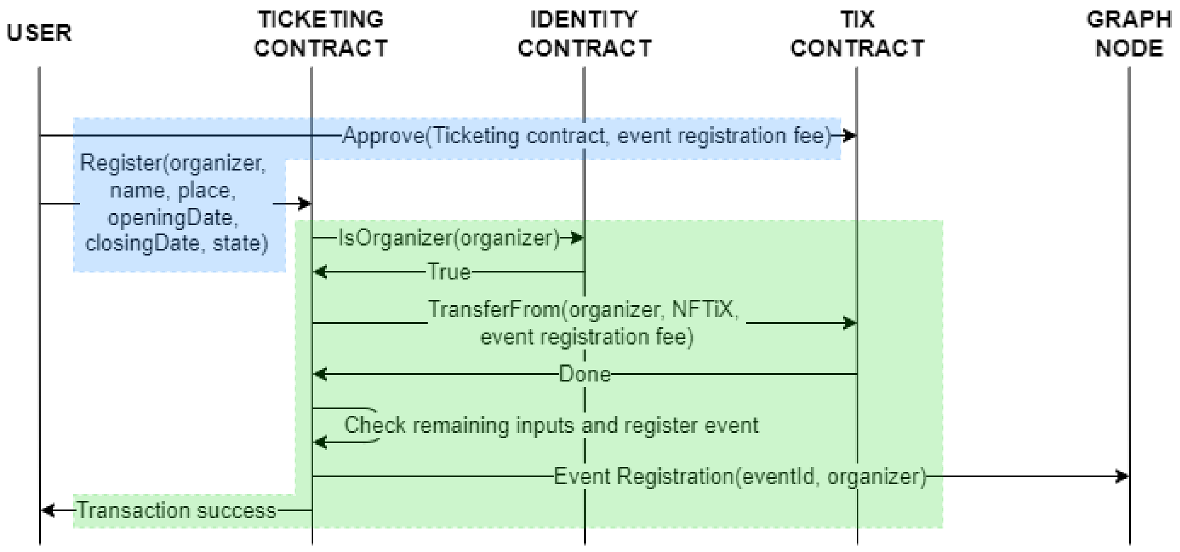

USER
TICKETING CONTRACT
IDENTITY CONTRACT
TIX CONTRACT
GRAPH NODE
Approve(Ticketing contract, event registration fee)
Register(organizer, name, place, openingDate, closingDate, state)
IsOrganizer(organizer)
True
TransferFrom(organizer, NFTiX, event registration fee)
Done
Check remaining inputs and register event
Event Registration(eventId, organizer)
Transaction success

Figure 4.3: Event registration

[2] https://tokenallowance.io/

An organizer can mint tickets for his events. To do this, it must provide an event ID that exists, a ticket ID, and an address of a registered spectator who will own the ticket. If the event ID is unknown, the organizer is not the owner of the event, or the address does not match a registered spectator, the operation is reverted. The organizer must also pay a minting fee for each minted ticket. The payment of the minting fee is analogous to the registration fee payment. The smart contract supports batch injections to make life easier for the organizer. The organizer must provide an event ID, a list of ticket IDs, and a list of addresses to do this. The figure 4.4 illustrates the flow of ticket minting.

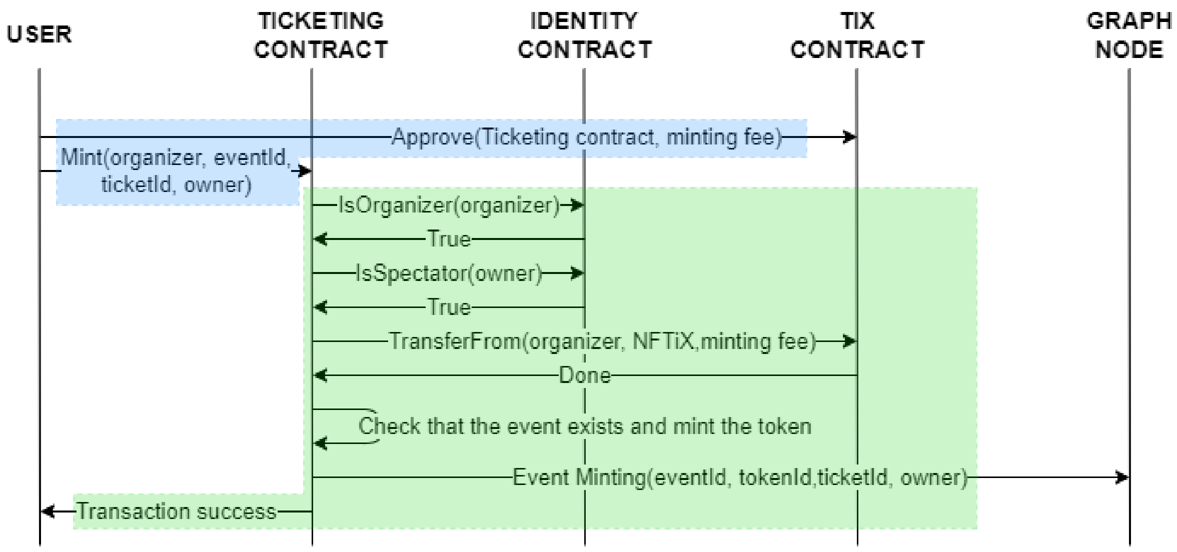

USER
TICKETING CONTRACT
IDENTITY CONTRACT
TIX CONTRACT
GRAPH NODE
Approve(Ticketing contract, minting fee)
Mint(organizer, eventId, ticketId, owner)
IsOrganizer(organizer)
True
IsSpectator(owner)
True
TransferFrom(organizer, NFTiX,minting fee)
Done
Check that the event exists and mint the token
Event Minting(eventId, tokenId,ticketId, owner)
Transaction success

Figure 4.4: Ticket minting

An organizer can burn or update the status of their tickets. This operation is analogous to minting and can also be executed by batch. However, it does not include any fees.

A spectator can transfer a ticket to another spectator. A transfer is successful if the transaction initiator is the ticket owner, the ticket recipient is a registered spectator, the event has not started yet, the event state is OPEN, and the ticket state is VALID. The figure 4.5 illustrates the flow of ticket transfer.

Finally, anyone can get all the information related to an event or a ticket by providing the specific event ID or token ID.

The above description presents the initial design. It works, but it can be improved. To pay the event registration and the minting fee, the organizer needs to approve the Ticketing contract to transfer his TIX to NFTiX. It means that the organizer must first do an approval transaction in the TIX contract and then an event registration transaction in the Ticketing contract. Doing two transactions is not ideal because the organizer has to pay twice the base gas fee. However, with the improvement we explain in the previous section, the organizer can create a TIX transfer approval proof to authorize the Ticketing contract to make the transfer on its behalf. With this approach, the organizer only needs to do a single transaction, and the event registration is now

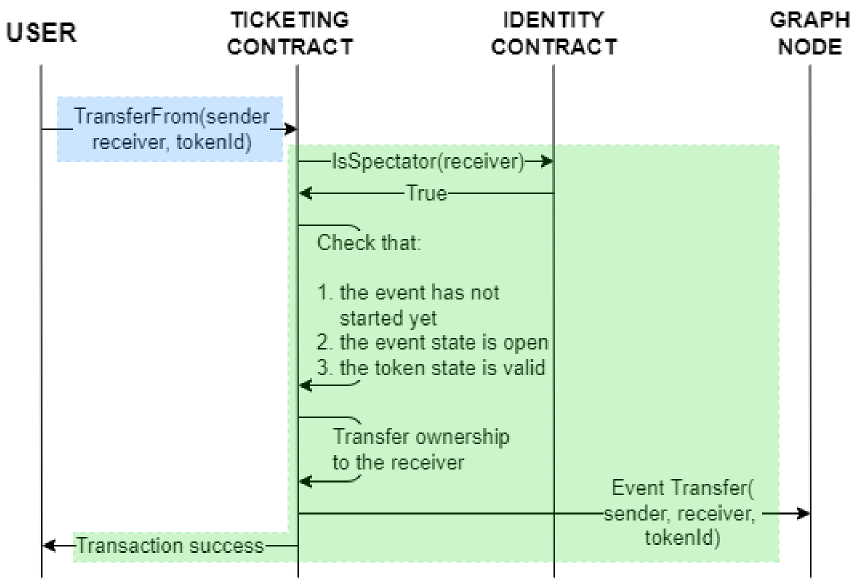

USER
TICKETING CONTRACT
IDENTITY CONTRACT
GRAPH NODE
TransferFrom(sender receiver, tokenId)
IsSpectator(receiver)
True
Check that:
1. the event has not started yet
2. the event state is open
3. the token state is valid
Transfer ownership to the receiver
Event Transfer(sender, receiver, tokenId)
Transaction success

Figure 4.5: Ticket transfer

atomic. We do the same for the payment of the minting fee.

We believe that the externally approved transfer is very useful because it reduces the number of transactions and allows the atomicity of complex tasks. Therefore, we also extend the ERC721 standard with this mechanism. This mechanism in the TIX contract and the Ticketing contract allow us to make atomic resale and swap. We detail this in the next section.

We would also like to mention an improvement that was ultimately not kept. A single contract manages the organizers' events and tickets with the current design. It could be helpful to have a dedicated ticketing contract per organizer so as not to mix data from different organizers and to have a specific on-chain logic per organizer. To do this, we can use a factory contract similar to the one used by UniswapV3. When an organizer registers an event on the factory, it automatically deploys a Ticketing smart contract dedicated to this event. We can manually deploy a Ticketing contract and then register it to the factory contract if we want specific business logic. We are not keeping this improvement because we aim to have a generic, simple, on-chain logic. For specific business logic, we prefer to use oracles and proof.

4.5.4 Exchange smart contract

The Exchange smart contract allows the spectators to resell and swap their tickets in complete safety. The contract lists all resale and swap offers. In addition, the contract verifies that the Approver has approved each transaction before executing it.

Let us start with ticket resale. The resale takes place in two stages. In the first stage, the seller creates a resale offer, and in the second stage, a buyer accepts the offer.

To create the resale offer, the seller must first approve the Exchange contract to operate its ticket. Then, he must make a transaction specifying which token he wants to resell and the price in USD. If he wishes, he can also specify an optional buyer. Only this person can accept the resale offer by specifying an optional buyer. If someone else tries to accept the offer, the transaction is reverted. The transaction succeeds if the following conditions are met: No resale offer exists for this token, no swap offer exists for this token, the initiator of the transaction is the token's owner, the price is not negative, the optional buyer, if defined, is a registered spectator, the resale approval proof is valid and signed by the Approver. Note that it is possible to cancel a resale offer as long as it has not been accepted yet.

To accept the resale offer, the buyer must make a transaction specifying which token he wishes to purchase and approve the contract to transfer an amount in TIX equivalent to the resale price. Note that the sale price is specified in USD, but the transaction is made in TIX. To determine how many TIXs need to be approved, the buyer requests the current resale price from the contract by providing the token ID of the ticket he wishes to purchase. The transaction succeeds if the following conditions are met: A resale offer exists for the token, the buyer is a registered spectator, the buyer is the optional buyer if specified, the approved amount is sufficient to satisfy the resale.

Note that a fee is taken on each resale. This fee is a percentage of the resale amount. The purpose of this fee is to cover the operating costs of NFTiX and give a commission to the event's organizer. The resale amount is split into three shares: the seller share, the NFTiX share, and the organizer share, respectively, 98%, 1% and 1% of the resale amount. The seller and the buyer pay each 50% of this fee. The figure 4.6 illustrates the flow of ticket resale (V1).

The swap works analogously to resale. The major difference is that a resale is an exchange of a ticket and money, whereas a swap is an exchange of two tickets.

To create a swap offer, spectator A must make a transaction specifying which token he wishes to swap and for which event he wishes to receive a token in exchange. As for the resale, he can specify an optional spectator B who is the only one who can accept the offer, and he must attach the swap approval proof signed by the Approver. The transaction succeeds if the following conditions are met: No resell offer exists for this token, no swap offer exists for this token, the event of the wanted ticket exists, spectator A is the token's owner, the optional spectator B, if defined, is registered, the swap approval proof is valid and signed by the Approver. It is also possible to cancel a swap offer if it has not been accepted yet.

To accept a swap offer, the spectator B must make a transaction specifying which offer he wishes to accept and which token he wishes to give in exchange. The transaction succeeds if the following conditions are met: A swap offer exists for token A, the spectator B is a registered spectator, the token provided by the spectator B is part of the desired event, the spectator B is the optional buyer if specified. The figure 4.7 illustrates the flow of ticket swap (V1).

The above description presents the initial design. It works, but we have the same problem as

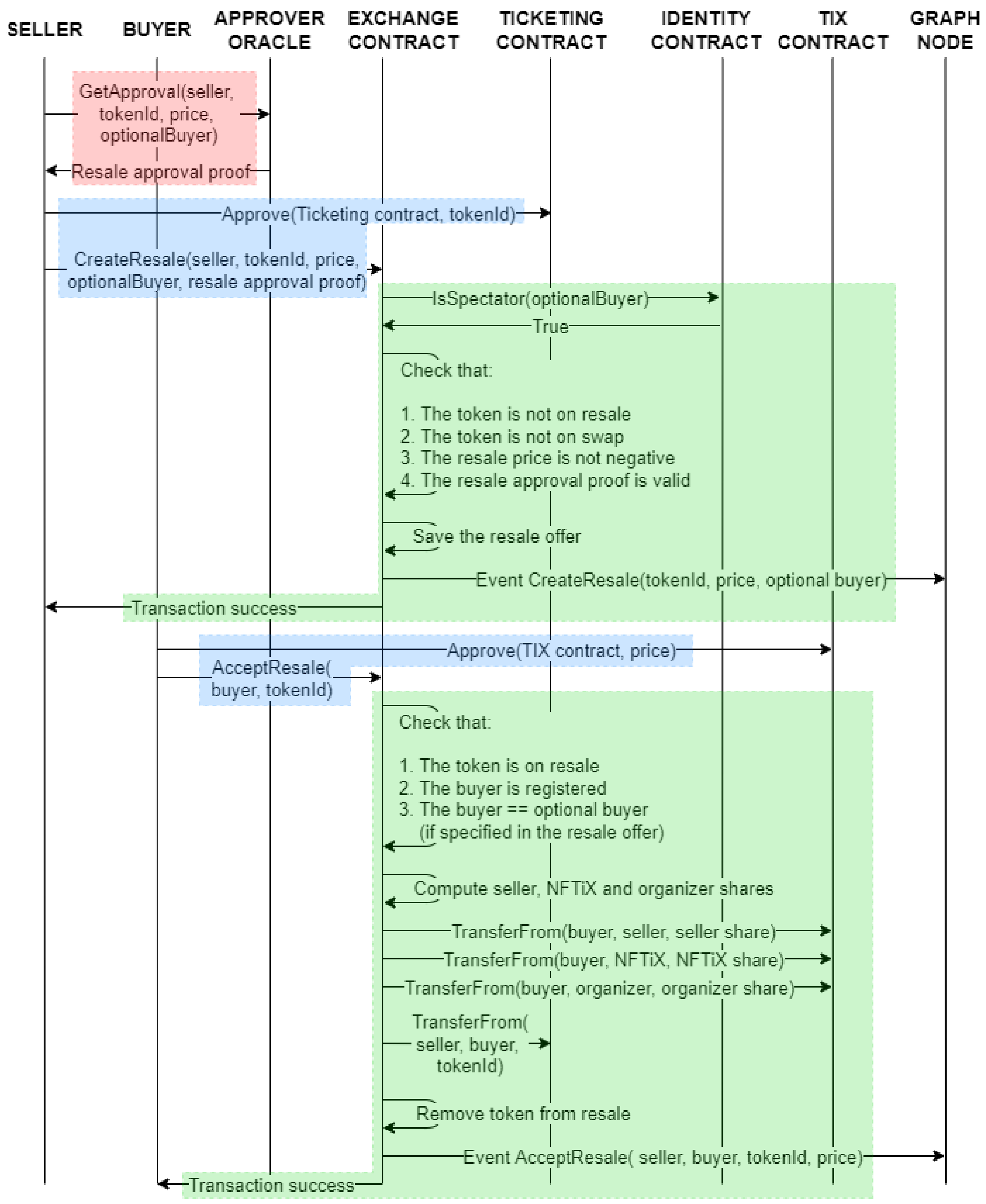

SELLER
BUYER
APPROVER ORACLE
EXCHANGE CONTRACT
TICKETING CONTRACT
IDENTITY CONTRACT
TIX CONTRACT
GRAPH NODE
GetApproval(seller, tokenId, price, optionalBuyer)
Resale approval proof
Approve(Ticketing contract, tokenId)
CreateResale(seller, tokenId, price, optionalBuyer, resale approval proof)
IsSpectator(optionalBuyer)
True
Check that:
1. The token is not on resale
2. The token is not on swap
3. The resale price is not negative
4. The resale approval proof is valid
Save the resale offer
Event CreateResale(tokenId, price, optional buyer)
Transaction success
Approve(TIX contract, price)
AcceptResale(buyer, tokenId)
Check that:
1. The token is on resale
2. The buyer is registered
3. The buyer == optional buyer (if specified in the resale offer)
Compute seller, NFTiX and organizer shares
TransferFrom(buyer, seller, seller share)
TransferFrom(buyer, NFTiX, NFTiX share)
TransferFrom(buyer, organizer, organizer share)
TransferFrom(seller, buyer, tokenId)
Remove token from resale
Event AcceptResale(seller, buyer, tokenId, price)
Transaction success

Figure 4.6: Ticket resale V1

before. When a spectator creates a resale offer, he must make two transactions. The first transaction authorizes the exchange contract to transfer his ticket. The second transaction creates the resale offer. The spectator who buys the ticket in resale must also make two transactions.

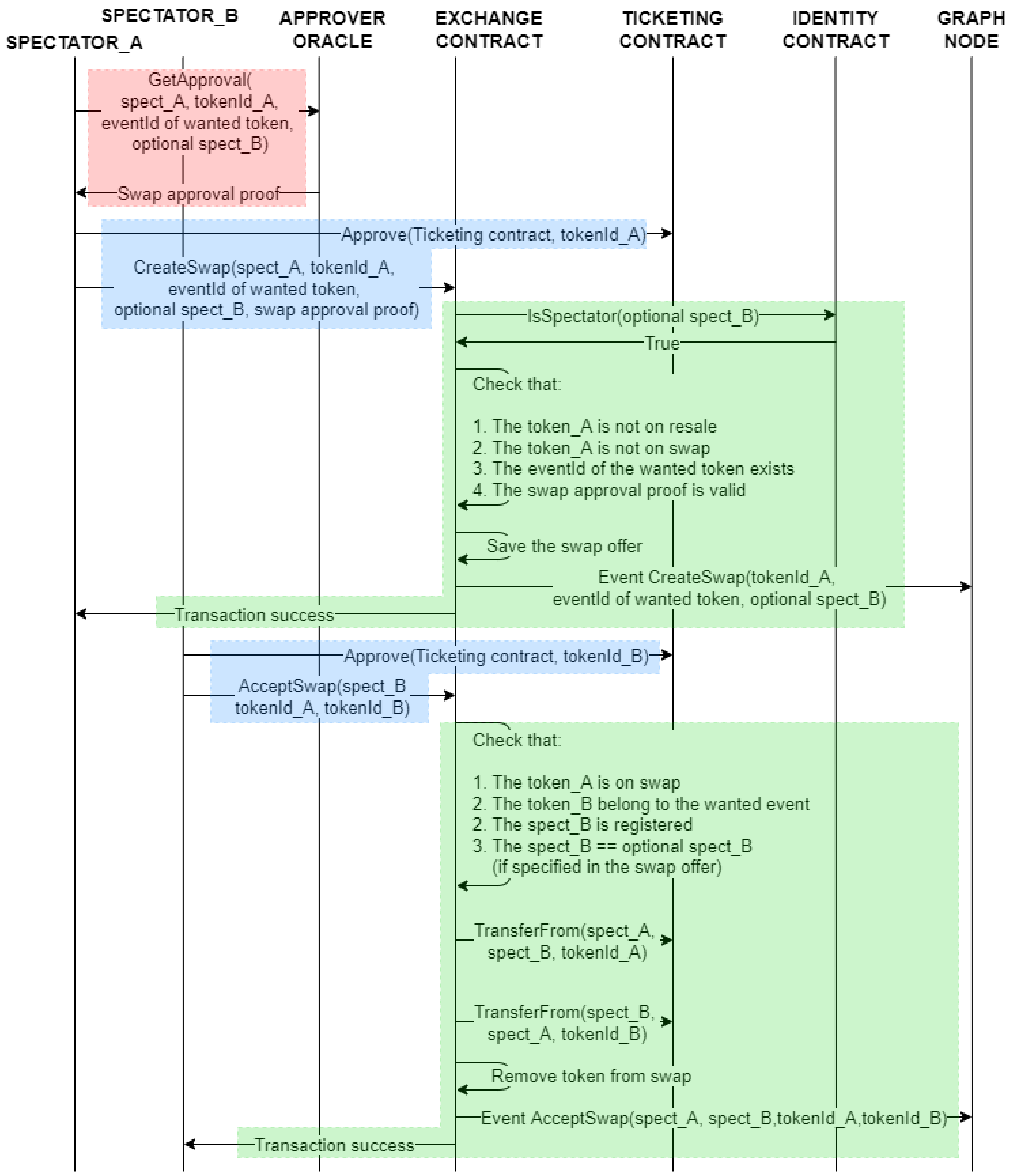

SPECTATOR_B
SPECTATOR_A
APPROVER ORACLE
EXCHANGE CONTRACT
TICKETING CONTRACT
IDENTITY CONTRACT
GRAPH NODE
GetApproval(spect_A, tokenId_A, eventId of wanted token, optional spect_B)
Swap approval proof
Approve(Ticketing contract, tokenId_A)
CreateSwap(spect_A, tokenId_A, eventId of wanted token, optional spect_B, swap approval proof)
IsSpectator(optional spect_B)
True
Check that:
1. The token_A is not on resale
2. The token_A is not on swap
3. The eventId of the wanted token exists
4. The swap approval proof is valid
Save the swap offer
Event CreateSwap(tokenId_A, eventId of wanted token, optional spect_B)
Transaction success
Approve(Ticketing contract, tokenId_B)
AcceptSwap(spect_B tokenId_A, tokenId_B)
Check that:
1. The token_A is on swap
2. The token_B belong to the wanted event
2. The spect_B is registered
3. The spect_B == optional spect_B (if specified in the swap offer)
TransferFrom(spect_A, spect_B, tokenId_A)
TransferFrom(spect_B, spect_A, tokenId_B)
Remove token from swap
Event AcceptSwap(spect_A, spect_B,tokenId_A,tokenId_B)
Transaction success

Figure 4.7: Ticket swap V1

The first gives the authorization to the Exchange contract to transfer his TIX and the second transaction accepts the resale offer.

As we explained earlier, this is due to how the ERC721 and ERC20 standards work. To transfer an asset, one must either be the owner or have received the approval of its owner.

In order to fix this limitation, our first approach was to have the TIX contract and ticketing contract whitelist the address of the Exchange smart contract so that it does not need to get user approval. However, this approach is not a good idea as it blurs the line between contracts and their role. Additionally, an attacker could steal TIX if the Exchange contract has a security hole. This approach does not respect the principle of least privilege.

Eventually, we found the solution we explained earlier, the externally approved transfer. The idea takes a little time to mature. In the following paragraphs, we explain how we came to the idea. We first understood that a transaction is just an operation signed by a user that changes the state of the blockchain. However, the user does not need to initiate this state change as long as the user approves the one operating. Therefore, the idea is to use cryptography so that spectators can create off-chain evidence that authorizes the Exchange contract to transfer TIX and tickets on their behalf. As explained previously, we have modified the ERC721 and ERC20 standards to accept this new type of approval. Note that with this approach, there is nothing new to build. We construct proof in the same way as those created by oracles. We also reused the verification mechanism used to verify the proof of the oracles to verify the proof issued by the users. As it is no longer necessary to approve the transfer of TIXs and tickets, we halve the number of transactions needed to carry out a resale.

We take the process even further by only needing a single transaction to complete a resale. A seller must complete a transaction that lists his resale offer on the Exchange contract with the current design. However, there is no need to list the offer directly on the contract. We can do that off-chain to avoid an additional transaction by letting the seller creates a resale offer proof. The proof is the same as the one issued by the Approver but signed by the seller. A buyer then uses this proof to purchase the tickets.

If we combine the two approaches, we offer the possibility of reselling a ticket by carrying out a single transaction! It is handy because it allows us to make atomic resales and thus do not oblige the buyer and the seller to trust each other. Note that the Exchange smart contract only executes a resale if all the proofs are present, and a party in possession of a TIX or ticket transfer approval proof cannot steal this asset. We do a thorough analysis in the security section.

Here is a high-level description of the flow. The seller asks the Approver to create a resale approval proof. Then, the seller creates two proofs. A resale offer proof that authorizes a buyer to purchase his ticket and a ticket transfer approval proof that authorizes the Exchange Contract to transfer the ticket on his behalf. The buyer creates three TIX transfer approval proofs, one per share, that authorize the Exchange contract to transfer the TIX on his behalf. Finally, the buyer creates a resale transaction that contains the six proofs and sends it to the blockchain. The smart contract checks that all authorizations are valid and executes the resale. The figure 4.8 illustrates the flow of ticket resale (V2).

We use the same approach for the swap in order to obtain an atomic swap. However, we also

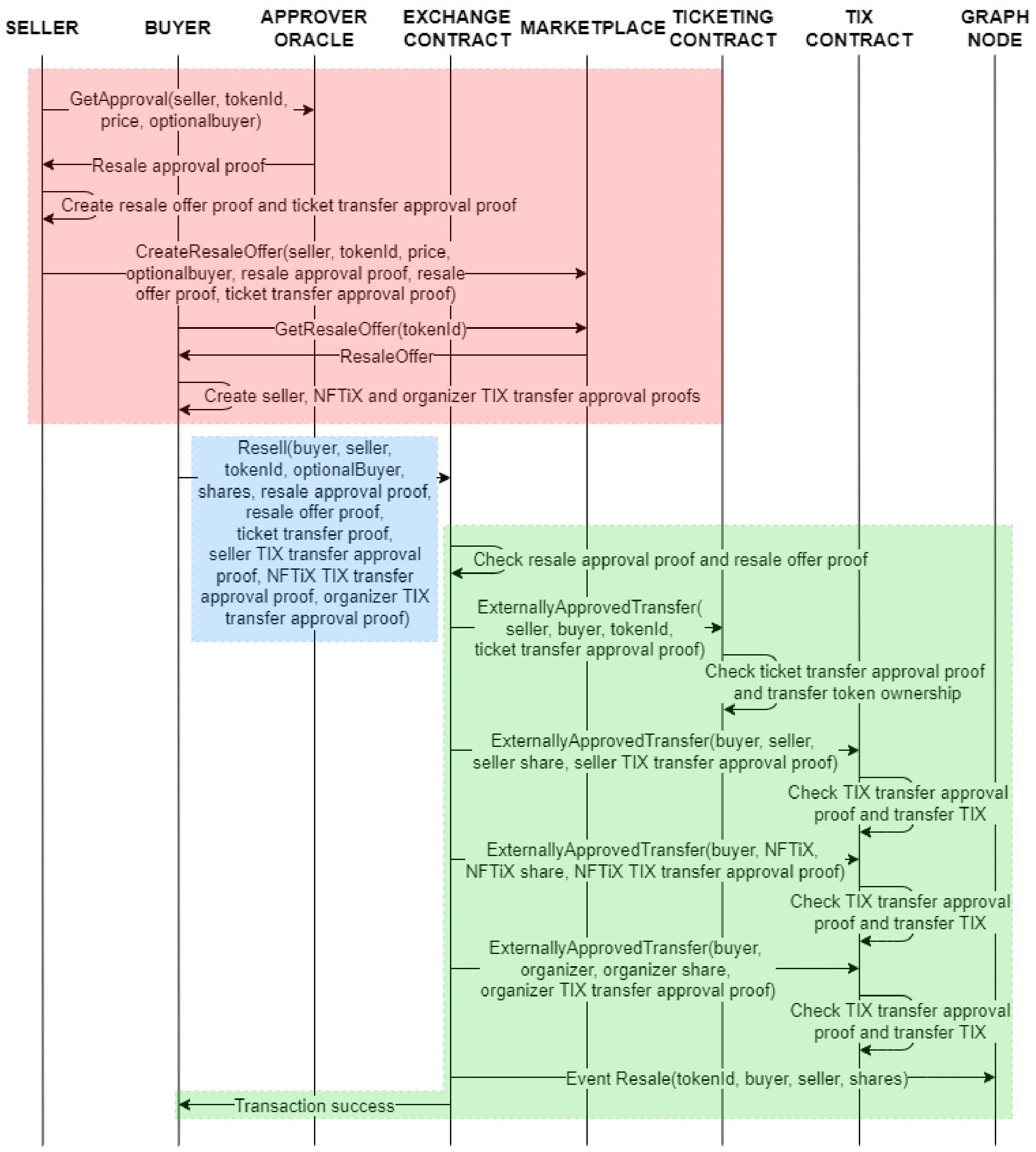

SELLER
BUYER
APPROVER ORACLE
EXCHANGE CONTRACT
MARKETPLACE
TICKETING CONTRACT
TIX CONTRACT
GRAPH NODE
GetApproval(seller, tokenId, price, optionalbuyer)
Resale approval proof
Create resale offer proof and ticket transfer approval proof
CreateResaleOffer(seller, tokenId, price, optionalbuyer, resale approval proof, resale offer proof, ticket transfer approval proof)
GetResaleOffer(tokenId)
ResaleOffer
Create seller, NFTiX and organizer TIX transfer approval proofs
Resell(buyer, seller, tokenId, optionalBuyer, shares, resale approval proof, resale offer proof, ticket transfer proof, seller TIX transfer approval proof, NFTiX TIX transfer approval proof, organizer TIX transfer approval proof)
Check resale approval proof and resale offer proof
ExternallyApprovedTransfer(seller, buyer, tokenId, ticket transfer approval proof)
Check ticket transfer approval proof and transfer token ownership
ExternallyApprovedTransfer(buyer, seller, seller share, seller TIX transfer approval proof)
Check TIX transfer approval proof and transfer TIX
ExternallyApprovedTransfer(buyer, NFTiX, NFTiX share, NFTiX TIX transfer approval proof)
Check TIX transfer approval proof and transfer TIX
ExternallyApprovedTransfer(buyer, organizer, organizer share, organizer TIX transfer approval proof)
Check TIX transfer approval proof and transfer TIX
Event Resale(tokenId, buyer, seller, shares)
Transaction success

Figure 4.8: Ticket resale V2

redefine its logic. Indeed, it does not make much sense that the initiator of the swap must specif for which event he wishes to receive a token in exchange. It is too vague and has no actual use case. A swap is a peer-to-peer operation. Two people have something the other wants, and they agree to trade it.

Let us describe how the swap works now. Both parties agree to swap their ticket. One of the two parties, let us call them Alice and Bob, asks the oracle to issue a swap approval proof by providing the address of Alice, the address of Bob, the token ID of Alice and the token ID of Bob. Alice and Bob both create two proofs. The first is a swap offer proof that attests they both agree to do the swap. The second is a ticket transfer approval proof that allows the Exchange contract to transfer the ownership of the tokens on their behalf. Finally, Alice or Bob makes a swap transaction containing the four proofs and sends it to the blockchain. The smart contract checks that all authorizations are valid and executes the swap. The figure 4.9 illustrates the flow of ticket swap (V2).

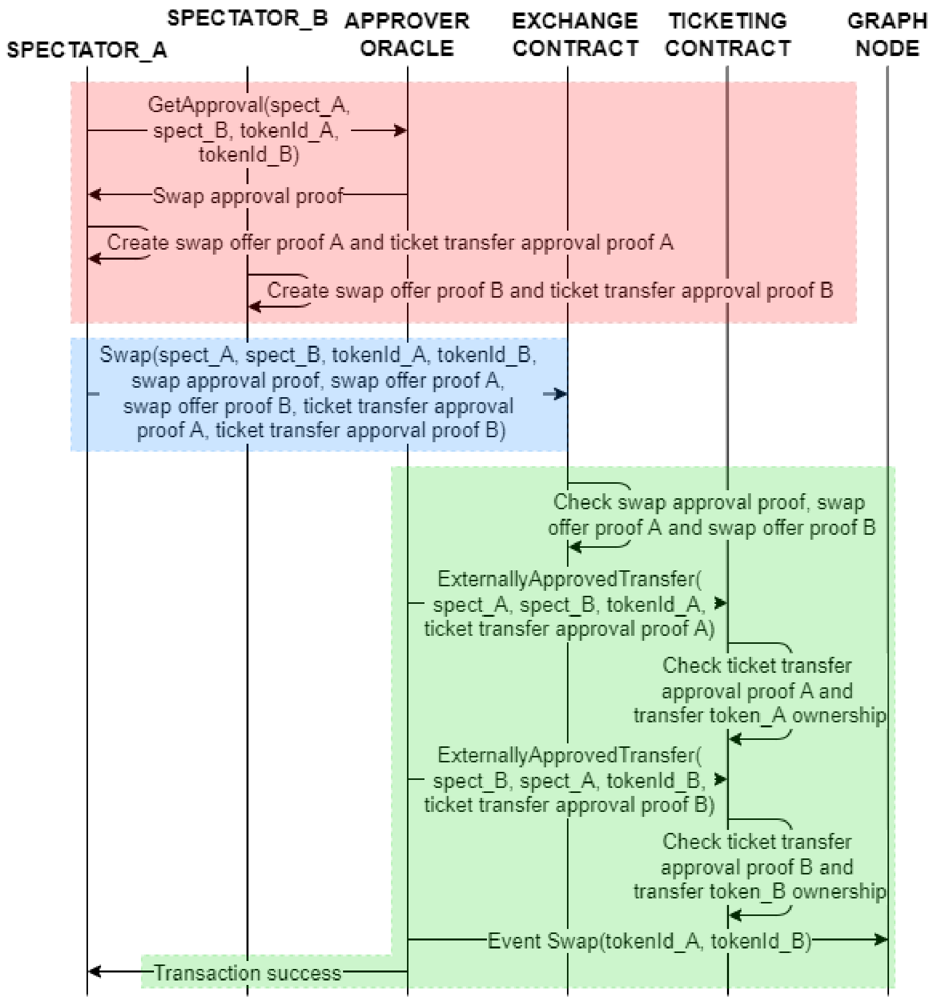

SPECTATOR_B
APPROVER ORACLE
EXCHANGE CONTRACT
TICKETING CONTRACT
GRAPH NODE
SPECTATOR_A
GetApproval(spect_A, spect_B, tokenId_A, tokenId_B)
Swap approval proof
Create swap offer proof A and ticket transfer approval proof A
Create swap offer proof B and ticket transfer approval proof B
Swap(spect_A, spect_B, tokenId_A, tokenId_B, swap approval proof, swap offer proof A, swap offer proof B, ticket transfer approval proof A, ticket transfer apporval proof B)
Check swap approval proof, swap offer proof A and swap offer proof B
ExternallyApprovedTransfer(spect_A, spect_B, tokenId_A, ticket transfer approval proof A)
Check ticket transfer approval proof A and transfer token_A ownership
ExternallyApprovedTransfer(spect_B, spect_A, tokenId_B, ticket transfer approval proof B)
Check ticket transfer approval proof B and transfer token_B ownership
Event Swap(tokenId_A, tokenId_B)
Transaction success

Figure 4.9: Ticket swap V2

To go even further in the process, we could also aggregate the proofs into one to reduce the complexity of verifying the proofs on the smart contract and, therefore, the transaction execution costs. For instance, we could aggregate the approval and offer proof because they are both checked by the Exchange contract. We could also aggregate the three externally approved TIX transfer proofs such that the TIX contract only need to verify a single proof. Of course, the TIX contract must also support batch transfer. Finally, note that we need a way to aggregate proofs such as homomorphic properties, which is not the case with the proofs we are using.

4.6 Graph node

The graph node runs the graph protocol and makes it possible to make complex queries that would be very difficult to do directly on the blockchain.

Let us understand why it is difficult to do complex queries on the blockchain. There are two main limitations. The first is the gas limit. Like transactions, calls cannot have a complexity exceeding a specific limit. If the size of the returned data is too large, the call fails. However, we can overcome this limitation if the data structure is simple. All we have to do is split a large query into several sub-queries. For example, if we want to obtain the balance of all the accounts of an ERC20 contract, we can query the first 100 balances in a first request, then the following 100 in a second request and so on. It requires two data structures on the smart contract, a list and a hashmap. The list stores the user's balance, and the hashmap maps the user's address to his balance's position in the list. This approach makes it possible to update a user's balance in O(1) because the hashmap tracks the position of his balance, and it is also possible to return all the balances by iterating over the list.

With this approach, it is also possible to easily create simple filters. Let us go back to the previous example and assume that the smart contract stores two additional hashmaps. The first hashmap tracks premium accounts, and the second hashmap tracks VIP accounts. If we want to get the balance of all premium and VIP accounts, we can iterate over the list of accounts and return only the accounts tracked by one of the two hashmaps.

However, this approach is limited because it is not possible to create queries as complex and rich as SQL would allow. There is no built-in primitive such as merge, join or filter to manipulate data. Therefore, if one wants to implement a complex query on a smart contract, he needs to implement by himself all these primitives. The function code would be very complicated to write, and the request would fail as soon as the number of data becomes too large because the maximum gas limit would be reached.

The graph overcomes this limitation by creating an index outside the blockchain. The index is built through events emitted by transactions and can be queried through graphql [3] queries.

Here is an overview of how indexing works. Let us use again the ERC20 contract storing the balance of users as an example. When a user transfers tokens to another user, he first creates a transaction with the following fields: his address, the recipient's address and the amount he wants to transfer. The user sends the transaction to the blockchain, which executes it. When the blockchain mines the transaction, it emits and logs an event containing the three preceding fields. The graph node listens to this event and updates its index by decrementing the sender's

[3]https://graphql.org/

balance and incrementing the receiver's balance. The figure 4.10 illustrates the flow of the graph node indexing (V2).

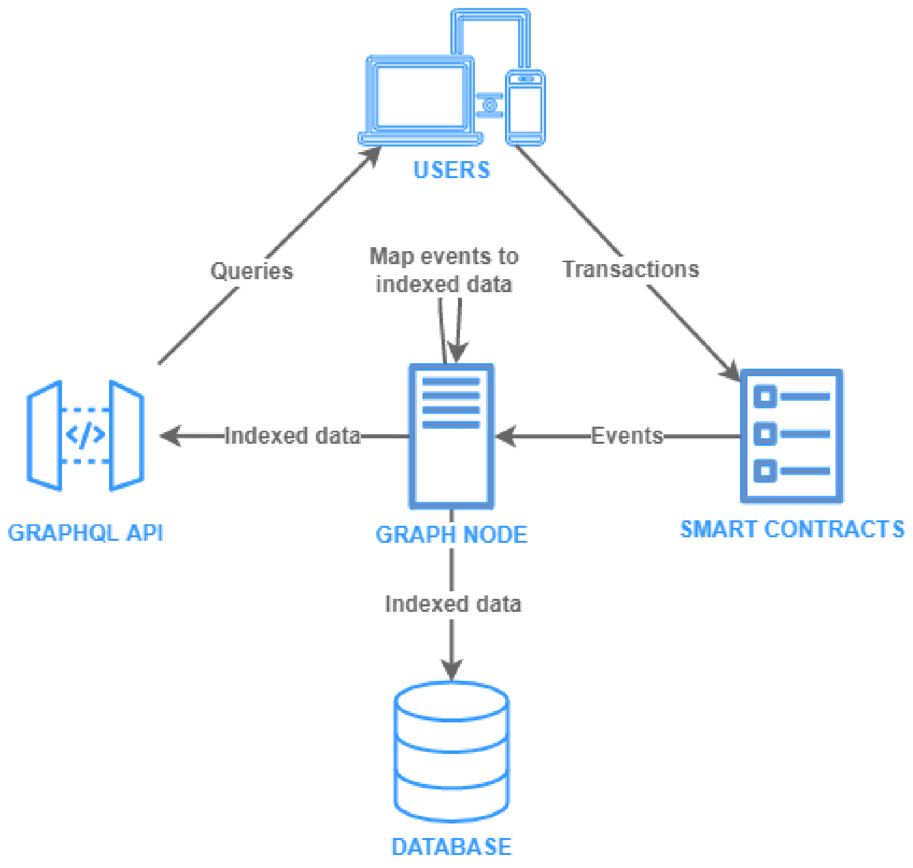


Figure 4.10: Graph node indexing

Now that we understand why it is hard to do complex queries or get bulk data, let us look at the indexes we use in our system. The graph node maintains three indexes. The first stores the user information, i.e. address, group and balance in TIX. The second stores the event information, i.e. event ID, organizer, name, location, opening date and time, closing date and time, state and number of tickets. The third stores the ticket information, i.e. token ID, ticket ID, event ID, owner and state. The graph node builds and maintains the three indexes by listening to the events sent by the four smart contracts. The list of events can be found in the Appendix A. Note that thanks to the externally approved transfer, we managed to reduce the amount of data stored in the indexes. With the initial design, we also needed to store the TIX allocation in the ticketing contract, the TIX allocation in the exchange contract, and the ticket status, i.e. void, on resale or on swap.

Chapter 5

Security

5.1 Security design

In this section, we explain a few points about the system's security and then list the assumptions on which our system is based.

5.1.1 Inheritance of Ethereum security

Note that the four smart contracts enforce the business logic on the blockchain. This particular point is essential because it allows the system to inherit the security of the blockchain as long as smart contracts do not have vulnerabilities.

The main benefit of running the business logic on the blockchain is that there is no single point of failure. Indeed, the blockchain is decentralized, i.e., managed by many different actors on different infrastructures and places. If one of the actors can no longer operate correctly, it is not a problem because the other actors continue to operate and provide the service. Furthermore, this property makes it difficult to control and manipulate the blockchain and our system. Indeed, to control the blockchain, it is necessary to control more than 51% of the actors, which is very difficult to do due to the decentralized and heterogeneous nature of the actors. Of course, zero risk does not exist, and it is crucial to keep in mind that the three main mining pools have together more than 51% of the hash rate. However, note also that Ethereum has never been compromised since its creation and is undoubtedly one of the most secure decentralized systems at the moment.

Another interesting point is that the implementation of smart contracts is public. Indeed, anyone can decompile a smart contract and check what it does. Therefore, anyone can verify that the contracts do what they are supposed to do and do not contain malicious code or vulnerabilities. It is an excellent thing because an open system brings trust to its users, and it forces Dapp developers to properly test their code and not rely on security by obfuscation.

Finally, the history of all user transactions is recorded on the blockchain and is unalterable. As soon as the blockchain mines a transaction, it logs it, and it is impossible to delete it (unless you control more than 51% of the network). Therefore, if anyone wants to attack the system, they must do so publicly. Sooner or later, the illegal operations will be noticed by monitoring tools or escalated by a user who notices a fraud, e.g. his tickets have disappeared.

5.1.2 Non-custodial design

We have chosen a non-custodial design, i.e. the keys of the Ethereum wallets are held by the users and not by the system, so that the users are really the owner of their asset and that only they can manipulate them. This approach distributes the keys over many actors, which increases the attack surface because there are more potential targets. However, it also drastically reduces the risk of a successful attack because to have a strong influence on the system, one must compromise a large number of keys.

5.1.3 Tamper-proof inputs

As we mentioned earlier, as much as possible of the business logic is on the blockchain so that it inherits its security. However, we need to use oracles to execute business logic and manipulate data that cannot be on the blockchain. In addition, spectators must create proofs of approval for certain operations. The approach we propose ensures the authenticity and integrity of the data sent on the blockchain because it relies on cryptography. Therefore, as long as the oracles and users are not compromised, all data entering the blockchain is secure, i.e. its integrity and authenticity are guaranteed.

5.1.4 Proof-based approval

The previous chapter explained that many operations used off-chain approval proofs to minimize the number of transactions required to complete it. We now need to show that this approach is secure. We can classify proofs into approval/offer proof and ticket/TIX transfer approval proof. Some operations only require one proof, while others require several. This analysis assumes no security flaw in smart contracts exists that would allow bypassing proof verification. Therefore, a transaction succeeds if and only if all the proofs are gathered and valid. Otherwise, the transaction fails. Also, note that as transactions are atomic, the system’s state is updated only in case of success.

Let us start with the TIX transfer approval proof used when registering a new event and minting new tokens. This proof authorizes the Ticketing contract to transfer TIX on behalf of the organizer. Someone with bad intentions can seek to use this proof to steal the organizer's TIX. To do this, he will try to modify the proof to change the recipient of the funds. In this case, he wants to replace the address of NFTiX with his own. However, this is not possible because the new message will not match the message with which the signature was created, and therefore the proof verification will fail. The only possible approach is finding a destination address that produces the same message as the NFTiX address when hashed. However, Keccak265 is second pre-image resistant. Therefore, it is in theory impossible to find an address different from that of NFTiX, which produces the same message. NFTiX may also seek to modify the amount to receive more TIX from the organizer. However, this is not possible for the same reasons explained above. Moreover, NFTiX has no reason to do this because it will alienate the organizer and make terrible publicity.

Now let us look at the resale that uses six proofs: a resale approval proof, a resale offer proof, a ticket transfer approval proof, and three TIX transfer proofs. As explained previously, it is not possible to exploit the TIX transfer approval proof in order to steal the TIX of the buyer. The same goes for the ticket transfer approval proof for the same reasons. There are three other possible attacks. The first is to make an illegal resale that the Approver has not approved. To do this, the attacker must successfully forge the signature of the Approver in order to create a fake resale approval proof. It is not possible because the attacker does not know the private key of the Approver. The second attack is to resell without the consent of the ticket owner. It is also not possible as the attacker does not have access to the ticket owner's private key, and therefore he cannot create a fake resale offer proof and ticket transfer approval proof. The third potential attack is to carry out a partial resale, such as the buyer receives the ticket but does not give the TIX in exchange or the seller receives the TIX but does not give the ticket in exchange. This attack cannot work because the operation is atomic. Finally, note that the same analysis is valid for the swap. We do not give it here to avoid lengthening this section unnecessarily.

To conclude our analysis, we will show that it is not possible to reuse old proofs to mount a replay attack and that it is also not possible to use a TIX transfer approval proof created by a third party to steal funds from him.

Note that no mechanism, such as a nonce, prevents reusing the same proof to mount a replay attack. The reason is that there is no need to have one because replay cannot succeed. Indeed, as the state is no longer the same, the checks performed by the smart contracts fail, so the transaction is reverted. Take, for example, the case of resale. First, a seller makes a resale offer which a buyer accepts. The ticket is transferred to the buyer, and the TIXs are transferred to the seller. Now, assume that the seller tries to steal TIXs by replaying the resale with the proofs that have been generated beforehand. The resale replay fails because the verification of the ticket owner performed by the smart contract fails. Indeed, the seller is no longer the owner of the ticket. Therefore, replay attacks do not work with our approach. Note that the same is true for the swap too. In the future work section, we describe a simple anti-replay mechanism that might be handy in specific situations.

Now let us look at the second attack, i.e., using third party proofs. This attack is not possible because the smart contracts check that the proofs are valid and that the transaction corresponds to the proof. Let us illustrate this with an example. Assume that a resale has occurred between a buyer and a seller; let us call them Alice and Bob. Now imagine that Alice resells a second ticket at the same price to another buyer; let us call her Clara. Clara tries to use the TIX transfer approval proof from the old resale to make a resale transaction while keeping her TIXs. This attack is not possible for two reasons. First, the proof verification fails because it was not signed by Clara but by Bob. Second, the Exchange contract verifies that the funds are indeed transferred from the buyer to the seller, i.e. from Clara to Alice, which is not the case. Therefore, it is not possible to mount this kind of attack.

The key point to understand is that the proofs no longer match the system's state once a transaction has been made. Therefore, although proofs do not contain a mechanism to prevent their reuse, it is impossible to use them for fraudulent purposes.

To conclude, we must specify that only smart contracts should receive this type of approval. If someone creates a TIX transfer approval proof for an external address, the owner can use this proof several times without anything preventing him from doing so. Therefore, he can completely steal the TIXs of the person who generated the proof. However, the anti-replay mechanism we explain in the future work section fix this problem.

5.2 Threat model

The design we propose is based on several assumptions. The first assumption is that the organizers and the operators of NFTiX are not malicious. Indeed, organizers can arbitrarily delete tickets, and system operators can revoke a user at will. Nothing prevents them from doing this kind of action. However, we assume that this assumption is reasonable as it is unlikely they will commit such actions. Indeed, they have more to lose than to gain. Their reputation and trust in the system would be damaged, directly impacting their income.

The second assumption is that the blockchain is considered completely secure and trustworthy. Thanks to the decentralized nature of the blockchain, this assumption is reasonable. Indeed, it is difficult to attack the blockchain. To manipulate it, one must take control of more than 51% of the network or break the cryptographic primitives it uses. It is highly unlikely that this will happen.

The third assumption is that the oracles' private key is not accessible by anyone. Therefore only oracles can generate valid proofs and update the price of the TIX. This assumption requires that the oracles and the infrastructure on which the oracles run are secure. It requires intensive testing and monitoring of this part of the system, which is crucial for having a regulated system.

We believe that this assumption is reasonable because the web as we know it is based on this same assumption.

Finally, we assume that the code of oracles and smart contracts does not contain vulnerabilities one can exploit to steal TIXs or tickets. Sadly, it is not possible to ensure that a code does not contain any flaws. However, we believe that due to the relatively low complexity of the system, it is possible to test the code sufficiently to have a high level of confidence in its security.

The design based on these four assumptions ensures that the business logic is applied as it should be and that users cannot deviate from the protocol. If they try, the system will deny their actions. For example, assume a malicious user seeks to make a resale that does not respect the resale rules. To perform this operation, he must obtain a resale approval proof from the oracle. As the resale is illegal, the oracle will refuse to provide the proof. The user cannot forge false proof because he does not have access to the private key of the oracle and because we assume that the cryptography is robust. The user cannot attack the smart contract because we assume it does not contain any vulnerability. The user cannot attack the blockchain either because we assume no one can manipulate it.

Finally, if a malicious user managed to find a flaw and exploit it, these actions would be visible because the blockchain records all the transactions. For example, suppose someone notices a vulnerability in a smart contract code that allows him to transfer any ticket, even if it does not belong to him. This flaw allows him to acquire all the tickets he wants. However, if he uses this flaw and transfers a ticket to himself, he automatically unmasks himself. In addition, it is possible to undo his actions. Indeed, the organizer can delete the stolen ticket and recreate a new one for the user who lost his ticket. Of course, it is also necessary to fix the flaw. Otherwise, the malicious user could start again.

Chapter 6

Implementation

The implementation of the system is on Github [1]. Note that the implementation is at the POC stage. The system is simulated through scripts and tests. However, there is everything to make it a production system with dedicated servers, APIs, etc.

The programming languages used are Solidity [2] for smart contracts and Typescript for scripts and tests.

We used Hardhat as a development and test environment with the following plugins: hardhat-waffle, ethereum-waffle, chai, hardhat-ethers ethers, hardhat-etherscan, dotenv, eslint, hardhat-gas-reporter, prettier, solhint, solidity -coverage, mocha, ts-node, typechain and typescript.

About the smart contract, we use the implementation of OpenZeppelin for the ERC20 standard, the ERC721 standard and the ECDSA signature verification.

[1] https://github.com/SirBouboute/marketplace
[2] https://docs.soliditylang.org/en/v0.8.12/

Chapter 7

Evaluation

7.1 Proof vs fact registry contract

We want to quickly compare our proof-based approval approach with another approach that more or less addresses the same problem. This approach is to use a smart contract to store facts [1]. This approach is more generic than ours because a fact is not limited to external approval. It can be anything like the price of a token against another one at a given time. It is suitable for building, for example, an AMM.

Another interesting point is that several parties can sign a fact. It could be convenient for NFTiX because, in the context of a resale, we could, for example, have a single fact signed by the seller, the buyer and the Approver instead of having six proofs. This approach also makes it possible to clearly separate the business logic and the external approval logic because the fact verification code is only on the fact registry contract and not on all the smart contracts as in our case.

However, we decided not to use this approach for several reasons. The first is that our approach is less gas-consuming because we only check the proofs but do not write anything to the blockchain. It is sometimes useful to have persistence, but it is not necessary in our case. The second reason is that a fact registry contract might be a single point of failure. If there is a vulnerability, our external approval system can be compromised entirely. By spreading the work across several different contracts, we reduce the risk of a vulnerability in one contract. The idea is to follow the least privilege principle. Finally, the main reason for our choice is that our approach allows doing atomic transactions and does not require preparation transactions such as approval or fact registration. We believe this is a real advantage because reducing the number of steps needed to complete an action dramatically reduces the system's attack surface and complexity while ensuring an equivalent level of security.

[1]https://docs.starkware.co/starkex-v4/starkex-deep-dive/smart-contracts-1/fact-registry

7.2 Performance

7.2.1 Experimental setup

The tests were performed on a Dell latitude 5501 laptop with an i7-9850H processor and 16GB of RAM in the WSL2 environment.

7.2.2 Throughput and latency

We cannot give an exact measurement of the throughput and latency on the blockchain because it would cost much gas, even on a testnet. Therefore, we give estimates based on statistics.

Ethereum operates at a rate of around 15 TPS and has a block confirmation time of around 15 seconds. It is insufficient to support NFTiX. Indeed, even by paying the maximum gas fee to integrate the transactions in the next block, the transaction confirmation time is a too long. Users do not want to wait 15 seconds for their transaction to complete. Moreover, there is another problem. We can reasonably estimate that NFTiX must support around 10,000 transactions daily. It represents the minting of 3'000'000 new tokens with a batch of size 300 or 10'000 resales. Ethereum has a daily volume of around 1,300,000 transactions. It means that NFTiX alone would use about 0.8% of the network, which is way too much. Therefore, Ethereum cannot support NFTiX at this time.

Polygon with around 100 TPS and a block confirmation time of around 2 seconds is a better candidate. A potential transaction confirmation time of around 2 seconds is perfectly acceptable. Moreover, using the same assumptions as before, we would use around 0.1% of the network. Solana is another interesting candidate. It has a TPS of around 2400 and a block confirmation time of around 0.8 seconds. NFTiX would use about 0.004% of the network, which is perfectly fine.

So, the estimation shows that it could be possible at the moment to deploy NFTiX on Polygon or Solana. However, as more and more projects are deployed on these networks, there is a high chance that in the future, they will suffer from the same congestion as Ethereum does. However, rollups seem very promising and would undoubtedly solve this scalability problem.

7.2.3 Max batch size

In the table 7.1 you can find the maximum batch size supported by the Ticketing contract on mint, burn and update operations on tickets. These are the maximum size of a batch before the operation is reverted because the gas limit is reached. We could use compression tricks and optimize the contract code to increase these maximum batch sizes, but we consider these numbers sufficiently good for a POC.

Max mint batch size	Max burn batch size	Max update batch size
300 ±10	970 ±10	410 ±10

Table 7.1: Maximum batch size per operation

7.2.4 Proof generation

In the table 7.2 you can see the time it takes to compute a hash, a signature, and a hash + a signature, i.e. a proof, of request batch of different sizes. The time is in seconds, and in parenthesis is the average time of a single operation.

Batch size	Hashing (s)	Signing (s)	Proof (s)
1'000	0.2164 (0.0002)	0.3566 (0.0004)	0.5779 (0.0006)
10'000	1.563 (0.0002)	2.968 (0.0003)	4.956 (0.0005)
100'000	14.59 (0.0001)	26.18 (0.0003)	50.37 (0.0005)
1'000'000	147.1 (0.0001)	256.7 (0.0003)	509.9 (0.0005)

Table 7.2: Proof batch generation time

The result highlights that creating the proofs is not time nor resource consuming. Note that the verification of the rules and the network latency are not considered in the measurement. Therefore, we expect the operation to take slightly more time to execute. However, note that these tests have been made on a single core of a laptop that is not the hardware that would be used in production. So we can conclude that the oracles can easily sustain more than 1500 proof generations per second, which is perfectly enough for our purpose.

7.3 Costs

The measure has been done the 21.02.2022 with the compiler version 0.8.12 and the optimizer enabled and set to 200.

7.3.1 Contract deployment

The table 7.3 gives the average prices for deploying smart contracts. Without surprise, it is costly to deploy them on Ethereum. However, it is very cheap to do it on Polygon, and it would be perfectly bearable for a company like Secutix SA.

Contract	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
Identity	763’699	126.70	0.03
TIX	1’447’729	240.18	0.06
Ticketing	3’873’386	642.59	0.16
ExchangeV1	2’179’969	361.66	0.09
ExchangeV2	1’255’857	208.35	0.05

Table 7.3: Contracts deployment price on Ethereum and Polygon

7.3.2 Transaction cost

The table 7.4, 7.5, 7.6, 7.7, 7.8 give the average prices of transactions. Note that we use the maximum batch size given above for batch operation. As for the deployment of contracts, it is not conceivable to operate NFTiX on Ethereum. Paying around 30 CHF to mint or resell a token is way too expensive. On the other hand, Polygon is very interesting because it only costs around 0.0004 CHF to mint a token. There is more data about costs on Ethereum and Polygon in the appendix B. There is also cost measurement for the Avalanche blockchain and the Binance smart chain.

Transaction	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
Register	54’663	9.07	< 0.01
Unregister	27’781	4.61	< 0.01
Revoke	30’582	5.07	< 0.01

Table 7.4: Identity contract transaction price on Ethereum and Polygon

Transaction	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
Mint	70’470	11.69	< 0.01
Burn	29’089	4.83	< 0.01
Buy	65’884	10.93	< 0.01
Sell	35’368	5.87	< 0.01
UpdateRates	31’841	5.28	< 0.01

Table 7.5: TIX contract transaction price on Ethereum and Polygon

7.3.3 Proof verification cost

Verification of proof on the blockchain costs between 6500 and 10000 gas. It corresponds to a cost increase between 7 and 15% compared to the transactions executed without proof verification. Although these numbers seem large, note that thanks to proofs, we can execute a good part of the business logic off-chain and drastically reduce the transaction costs.

Transaction	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
RegisterEvent	213’381	35.40	0.01
UpdateEventState	31’032	5.15	< 0.01
Mint	173’709	28.82	0.01
MintBatch	28’984’724	4808.55	1.18
Burn	29’089	4.83	< 0.01
BurnBatch	7’243’121	1201.63	0.30
UpdateTokenState	35’445	5.88	< 0.01
UpdateTokenStateBatch	3’812’565	632.50	0.16
TransferFrom	72’141	11.97	< 0.01

Table 7.6: Ticketing contract transaction cost

Transaction	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
CreateResale	83’772	13.90	< 0.01
AcceptResale	157’745	26.17	0.01
CancelResale	30’765	5.10	< 0.01
CreateSwap	73’467	12.19	< 0.01
AcceptSwap	87’081	14.45	< 0.01
CancelSwap	30’826	5.11	< 0.01

Table 7.7: Exchange V1 contract transaction price on Ethereum and Polygon

Transaction	Avg gas (Gwei)	Avg cost in Ethereum (CHF)	Avg cost in Ethereum (CHF)
Resell	186’680	30.97	0.01
Swap	111’984	18.58	< 0.01

Table 7.8: Exchange V2 contract transaction price on Ethereum and Polygon

Chapter 8

Limitation and future work

8.1 Off-chain ticket state

When designing the system, we tried to put the business logic on-chain as much as possible. The state of a ticket is one of the elements managed on-chain. However, upon reflection, we think this is a mistake for the following reasons. The INVALID ticket state is unnecessary because it is possible to delete no longer needed tickets. In addition, it costs less to delete a ticket than to update its state, thanks to the gas refund when we free a memory element. The SCANNED state is useful but only locally for access control. Therefore, it is not necessary to store it on the blockchain. In addition, updating each scanned ticket can potentially create heavy congestion on the blockchain if the access control updates several thousand states simultaneously. It can be problematic as tickets can be scanned multiple times before updating their state. Therefore, this information should be stored off-chain and updated in batch if needed on the blockchain.

8.2 Proof of ownership and scanner oracle

This thesis has not addressed the problem of QR code management. TIXnGO encrypts QR codes and reveals them before the event start. There are two ways to reveal the QR code. The first method is through Bluetooth beacons that send a signal. When the user is nearby, the phone reveals the QR code. The second method is time-based. A few minutes before the gate opens, the phone reveals the QR code.

The problem with these approaches is that the QR code and the decryption key must be permanently stored on the user's phone because the user cannot necessarily access the internet. Although it is possible to store the private key in the phone's secure enclave and extensively test the application code to ensure it is challenging to extract the QR code and use it for malicious purposes, this approach is not optimal.

Another approach is to generate a QR code that depends on the ticket owner, i.e. if two people had the same ticket, they would see a different QR code. With this approach, it is not necessary to encrypt the QR code. This approach is relatively easy to set up and has already been implemented and tested by OpenSea [1]. The idea is to generate a proof of ownership attesting that the person is the ticket holder. To do this, the person proves that he is the owner of the address holding the ticket. The proof is then used to construct the QR code.

8.3 External payment channel and proof

Currently, crypto remains a niche market, and most people still prefer to pay with fiat currencies—however, the TIX token powers our system. There are two ways to solve this problem. The first is to use a fiat to crypto converter such as Ramp [2]. In the context of a resale, a user can use Ramp to purchase the number of TIXs required to satisfy the resale with his credit card.

However, this approach implies that the seller receives TIXs when he may wish to receive the amount in fiat currency. It is possible to convert TIX into fiat currency, but this involves conversion and additional costs. To fix this limitation, NFTiX can also support an external payment system such as Stripe [3]. This approach requires a slight change in how the resale works. We replace the three buyer's TIX transfer approval proofs with a single proof issued by an oracle that handles external payments. First, the oracle creates a payment request. Once the buyer completes the payment, the oracle generates proof that attests that the payment has been executed and returns it to the buyer who uses it to make the resale.

8.4 Simple replay mechanism

The security section shows that our design does not need a replay mechanism. However, it could be handy to have a replay mechanism in other situations.

We can build a simple replay mechanism by maintaining a nonce on-chain. When a user or an oracle create a proof, it uses the current nonce. When smart contracts verify proof, they increase the nonce. The transactions are still atomic, and the additional cost is marginal. This approach should work just fine for the user. However, it could be inconvenient for the oracle because they cannot generate proof in parallel. The smart contract increments the nonce sequentially when verifying the proof, and precedent transactions may fail. Suppose there are two proofs, A and B,

[1]https://medium.com/opensea/cryptotickets-the-first-blockchain-based-tickets-are-live-on-opensea-5a29d0223c3d

[2]https://ramp.network/

[3]https://stripe.com/

and two transactions, A and B, using proof A and B, respectively. If transaction A fails, transaction B will fail too because the nonce has not been incremented by transaction A. However, we could use the nonce of the transaction to construct the proof. This way, even if transaction A is reverted, transaction B does not fail because the transaction nonce is incremented even if the transaction fails.

8.5 Decentralized oracles

As we mentioned previously, oracles are centralized. It is not a good thing because it creates a single point of failure, opens the door to censorship, and goes against the philosophy of decentralization. We could use a decentralized oracle such as Chainlink to overcome these problems.

8.6 Payment splitter with single proof

With our approach, the buyer of a ticket needs to create three TIX transfer approval proof, and the Exchange smart contract needs to call three times the transferFrom method of the TIX contract to transfer the three shares. We can use a payment splitter and a single proof to avoid this. The splitter checks that the proof is valid and then transfers the shares in a single function call to save some gas.

8.7 TIX token features

For the moment, the TIX token has very little use, and ETH could perfectly replace it. Here are some ideas that might make it more useful.

The TIX could be used as a governance token to vote for new features in the application. For example, it would be possible to vote to determine if the spectators would like to have the possibility of claiming a souvenir ticket of an event in the form of NFT. The governance could also let the organizers vote for the future features they would like the development team to implement first.

It would also be possible to create a reward pool that can be shared regularly between the most assiduous users of the application or people who discover security flaws in the system. We can fill the pool by taking a small commission on each resale of tickets.

Another cool feature would be that the exchange redeems tickets for TIXs. Sometimes someone cannot attend an event for some reason and fails to sell their ticket. It would be great to let that person returns his ticket and get TIXs in exchange because he could then use them to buy another ticket in the future. It could create a kind of virtuous circle.

Finally, the token should be listed in a decentralized exchange such as Uniswap or Curve instead of the buy function that locks ETH and mint TIX in exchange. This approach is not scalable because if the TIX takes much value compared to ETH, the amount of ETH in the smart contract will no longer be enough to reimburse people who sell their TIX.

8.8 Performance test on zkSync and StarkNet

The zk rollups, which in our opinion are the future of blockchain scalability, are becoming increasingly popular. It would be interesting to see how well the system would perform and what we would need to modify in the system in order to make it work on these layers 2.

8.9 Mixing custodial and non-custodial design

As we explained previously, a non-custodial design is necessary to ensure that the user owns his assets. However, it is also risky because users are often unaware of the risks. On the other hand, a custodial design is safer because the system uses high-security standards to protect the private key.

Why not mix the two worlds to have a high level of security while allowing the users to own their assets? It could be possible with an MPC. The idea is that both the user and the system provide a share of the transaction signature. With this approach, the user still controls his asset because the transaction signature cannot be created if he does not provide his share. A high level of security is achieved because the system uses the highest security standard to protect its key.

In addition, this approach could perhaps make it possible to build a regulation system similar to the one we propose but much simpler because it does not require on-chain proof verification. Indeed, in addition to the user and system share, the regulator could bring its share to give its approval. If the regulator does not give its share, the transaction is refused. However, we must also design a mechanism to ensure that the signature includes the shares of all parties. Adapting Shamir's Secret Sharing could be a good starting point to build such a mechanism.

Chapter 9

Conclusion

NFTiX is a decentralized and non-custodial ticket engine that aims to power the new TIXnGO ticket distribution system and marketplace. It includes a user management system, an event and ticket management system, and a ticket exchange that allows secure and regulated ticket exchanges between untrusted parties. To our knowledge, there is no such system yet at the time of writing this thesis.

Our security analysis shows that the system fulfils well its role in the fight against the black market and fraud. Our evaluation shows that it is possible to put NFTiX into production because the costs would not be too high, and the performance would be sufficient.

Finally, our core contribution is the proof-based strategy that allows decentralized and non-custodial exchange regulation, the new approval mechanism for the ERC20 and ERC721 standards allowing atomic execution of complex tasks and fixing a well-known vulnerability in the approval mechanism, and our utterly non-custodial ticket engine allowing users to be the sole owners of their assets.

Bibliography

[1] Zak Ali. *Global Online Event Ticketing Market to Reach 60 Billion by 2026.* URL: https://www.prnewswire.com/news-releases/global-online-event-ticketing-market-to-reach-60-billion-by-2026-301394955.html (visited on Jan. 25, 2022).

[2] *TIXnGO.* URL: https://www.tixngo.io/ (visited on Jan. 25, 2022).

[3] *Blockchain.* URL: https://fr.wikipedia.org/wiki/Blockchain (visited on Jan. 25, 2022).

[4] *Decentralized application.* URL: https://en.wikipedia.org/wiki/Decentralized_application (visited on Jan. 25, 2022).

[5] *Decentralized finance.* URL: https://en.wikipedia.org/wiki/Decentralized_finance (visited on Jan. 25, 2022).

[6] *Ethereum.* URL: https://ethereum.org/en/ (visited on Jan. 25, 2022).

[7] Ethereum Classic Cooperative. *Keccak256.* URL: https://medium.com/etccooperative/why-change-the-proof-of-work-algorithm-to-keccak-256-sha3-e327b8313824 (visited on Jan. 26, 2022).

[8] *Chainlink.* URL: https://chain.link/ (visited on Jan. 26, 2022).

[9] Eth. *Distinction between calls and transactions.* URL: https://ethereum.stackexchange.com/a/770 (visited on Feb. 4, 2022).

[10] *SHA3.* URL: https://en.wikipedia.org/wiki/SHA-3 (visited on Feb. 4, 2022).

[11] *The Graph.* URL: https://thegraph.com/docs/en/ (visited on Feb. 4, 2022).

[12] *Token allowance.* URL: https://tokenallowance.io/ (visited on Feb. 7, 2022).

[13] *GraphQL.* URL: https://graphql.org/ (visited on Feb. 8, 2022).

[14] *Smart contract event and logs in Ethereum.* URL: https://consensys.net/blog/developers/guide-to-events-and-logs-in-ethereum-smart-contracts/ (visited on Feb. 8, 2022).

[15] *Proof of ownership.* URL: https://medium.com/opensea/cryptotickets-the-first-blockchain-based-tickets-are-live-on-opensea-5a29d0223c3d (visited on Feb. 11, 2022).

[16] *Taurus.* URL: https://www.taurushq.com (visited on Feb. 14, 2022).

[17] *Taurus ggba switzerland.* URL: https://www.ggba-switzerland.ch/en/geneva-based-fintech-taurus-launches-the-worlds-first-regulated-digital-assets-marketplace/ (visited on Feb. 14, 2022).

[18] *FINMA securities law.* URL: https://www.finma.ch/en/authorisation/banks-and-securities-firms/getting-licensed/securities-firms/ (visited on Feb. 14, 2022).

[19] *TDX.* URL: https://t-dx.com/ (visited on Feb. 14, 2022).

[20] *New swiss DLT regulation.* URL: https://www.caplaw.ch/2021/new-swiss-dlt-regulation-status-update-and-outlook/ (visited on Feb. 14, 2022).

[21] *GET protocol.* URL: https://docs.get-protocol.io/ (visited on Feb. 14, 2022).

[22] DEDIS lab. *Omniledger.* URL: https://eprint.iacr.org/2017/406.pdf (visited on Feb. 14, 2022).

[23] Starkware. *Fact registry.* URL: https://medium.com/starkware/the-fact-registry-a64aafb598b6 (visited on Feb. 23, 2022).

[24] Starkware. *Fact registry documentation.* URL: https://docs.starkware.co/starkex-v4/starkex-deep-dive/smart-contracts-1/fact-registry (visited on Feb. 23, 2022).

[25] *Attack on approve and transfer method.* URL: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbTOmooh4DYKjA_jp-RLM/edit (visited on Feb. 23, 2022).

[26] Jeremy Clark Reza Rahimian. *TokenHook: Secure ERC-20 smart contract.* URL: https://arxiv.org/pdf/2107.02997.pdf (visited on Feb. 23, 2022).

Appendix A

List of events

The structure of an event is the following:

```
EventName(typeOfFirstElt nameOfFirstElt, typeOfSecondElt nameOfSecondElt, ...)
```

Identity

```
Registation(address user, bytes32 class, bytes32 hash, bytes signature);
Unregistration(address user);
Revocation(address user);
```

TIX

```
Purchase(address user, uint256 amount);
Sale(address user, uint256 amount);
Transfer(address from, address to, uint256 value);
Approval(address owner, address spender, uint256 value);
```

Ticketing

```
EventRegistration(uint256 eventId, address owner);
EventUpdate(uint256 eventId, bytes32 state);
Minting(uint256 tokenId, uint256 ticketId, address owner);
Burning(uint256 tokenId, uint256 ticketId);
TokenUpdate(uint256 tokenId, bytes32 state);
```

```
Transfer(address from, address to, uint256 tokenId);
Approval(address owner, address approved, uint256 tokenId);
```

Exchange V1

```
CreateResale(uint256 tokenId, uint256 price, address optionalBuyer);
CancelResale(uint256 tokenId);
AcceptResale(address seller, address buyer, uint256 tokenId, uint256 price);
CreateSwap(uint256 tokenId, uint256 eventIdOfWantedToken, address optionalParticipant);
CancelSwap(uint256 tokenId);
AcceptSwap(address creator, address acceptor, uint256 creatorTokenId,
           uint256 acceptorTokenId);
```

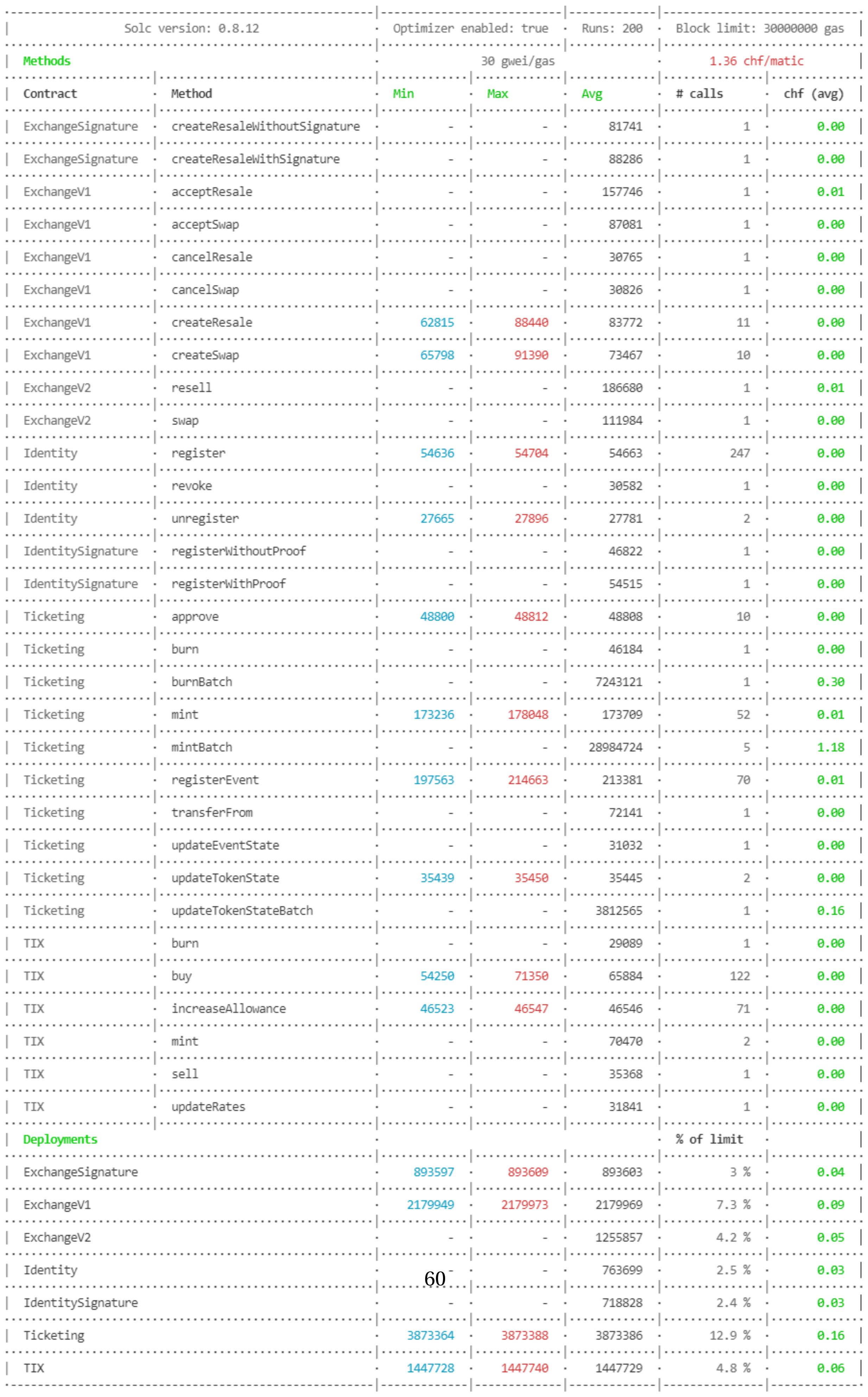
Exchange V2

```
Resale(uint256 tokenId, address buyer, address seller, address tixngo, address
       organizer, uint256 sellerShare, uint256 tixngoShare, uint256 organizerShare);
Swap(uint256 tokenId_A, uint256 tokenId_B, address user_A, address user_B);
```

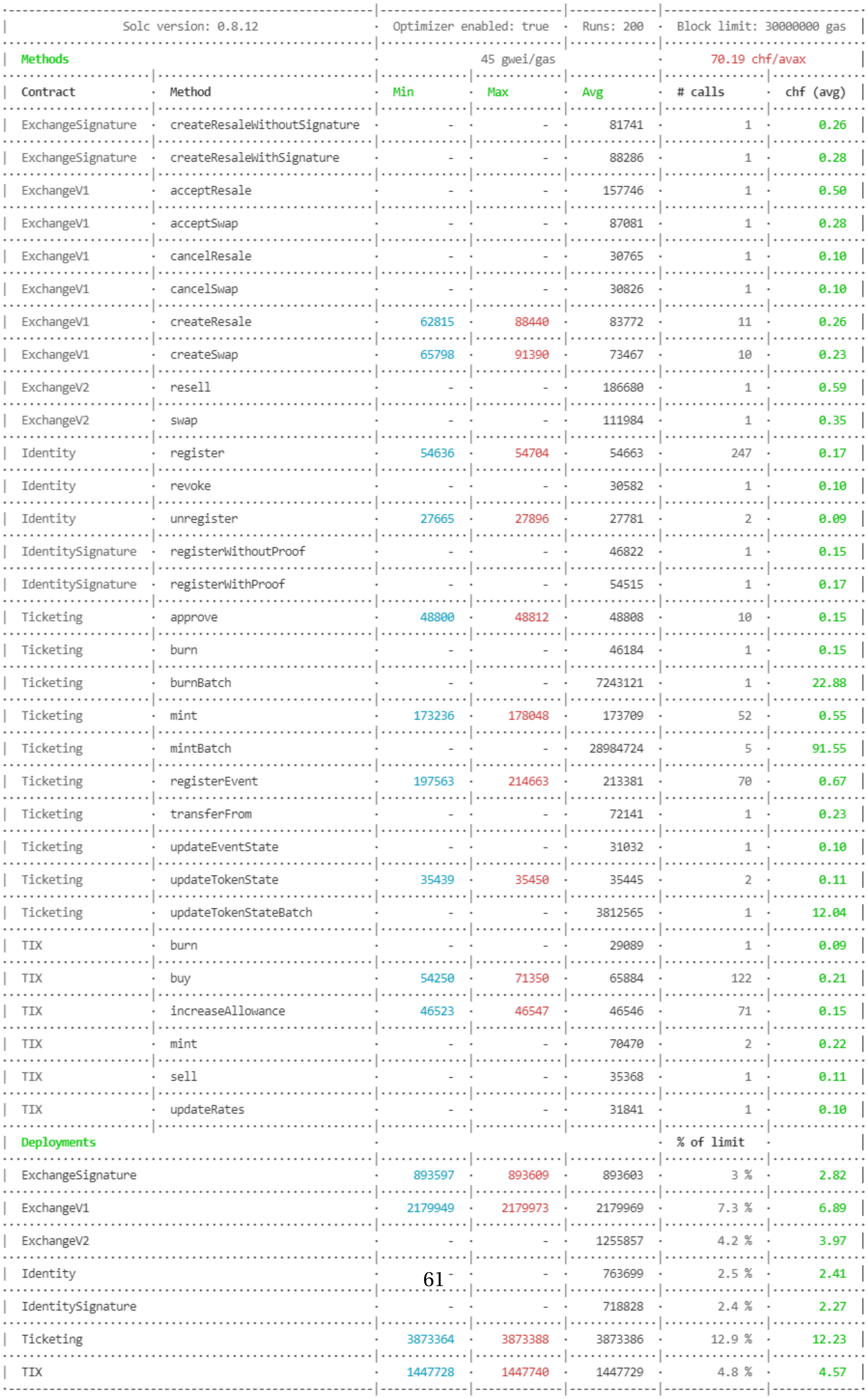
Appendix B

Extended cost

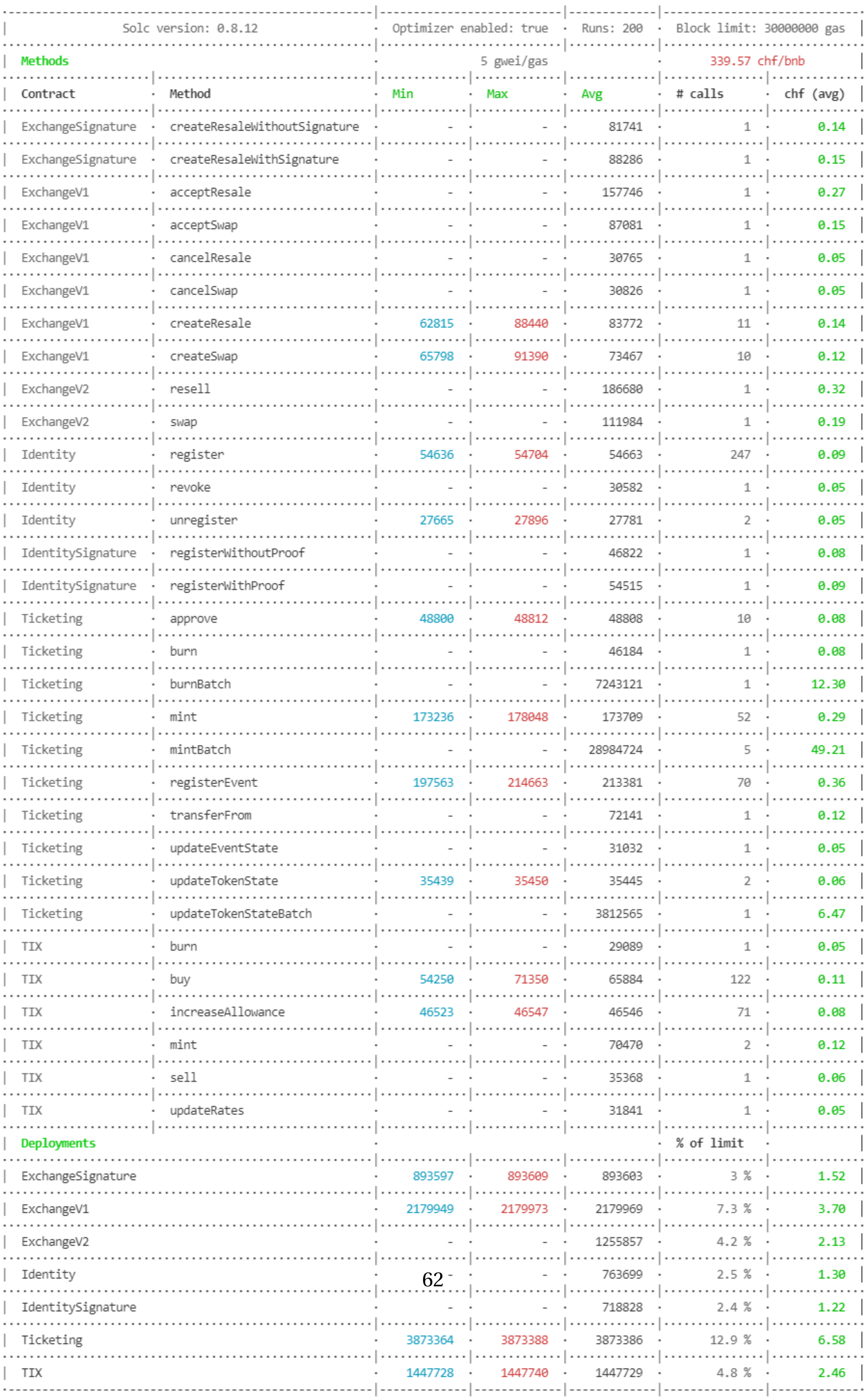
```
·-----------------------------------------------|---------------------------|-------------|-----------------------------·
|             Solc version: 0.8.12              ·  Optimizer enabled: true  ·  Runs: 200  ·  Block limit: 30000000 gas  │
················································|···························|·············|······························
|  Methods                                      ·        69 gwei/gas        ·       2404.34 chf/eth       │
····················|···························|·············|·············|·············|···············|··············
|  Contract         ·  Method                   ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithoutSignature  ·  -  ·  -  ·  81741  ·  1  ·  13.56  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithSignature     ·  -  ·  -  ·  88286  ·  1  ·  14.65  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptResale                  ·  -  ·  -  ·  157746  ·  1  ·  26.17  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptSwap                    ·  -  ·  -  ·  87081  ·  1  ·  14.45  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelResale                  ·  -  ·  -  ·  30765  ·  1  ·  5.10  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelSwap                    ·  -  ·  -  ·  30826  ·  1  ·  5.11  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createResale                  ·  62815  ·  88440  ·  83772  ·  11  ·  13.90  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createSwap                    ·  65798  ·  91390  ·  73467  ·  10  ·  12.19  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV2         ·  resell                        ·  -  ·  -  ·  186680  ·  1  ·  30.97  │
····················|···························|·············|·············|·············|···············|··············
|  ExchangeV2         ·  swap                          ·  -  ·  -  ·  111984  ·  1  ·  18.58  │
····················|···························|·············|·············|·············|···············|··············
|  Identity           ·  register                      ·  54636  ·  54704  ·  54663  ·  247  ·  9.07  │
····················|···························|·············|·············|·············|···············|··············
|  Identity           ·  revoke                        ·  -  ·  -  ·  30582  ·  1  ·  5.07  │
····················|···························|·············|·············|·············|···············|··············
|  Identity           ·  unregister                    ·  27665  ·  27896  ·  27781  ·  2  ·  4.61  │
····················|···························|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithoutProof          ·  -  ·  -  ·  46822  ·  1  ·  7.77  │
····················|···························|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithProof             ·  -  ·  -  ·  54515  ·  1  ·  9.04  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  approve                       ·  48800  ·  48812  ·  48808  ·  10  ·  8.10  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  burn                          ·  -  ·  -  ·  46184  ·  1  ·  7.66  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  burnBatch                     ·  -  ·  -  ·  7243121  ·  1  ·  1201.63  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  mint                          ·  173236  ·  178048  ·  173709  ·  52  ·  28.82  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  mintBatch                     ·  -  ·  -  ·  28984724  ·  5  ·  4808.55  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  registerEvent                 ·  197563  ·  214663  ·  213381  ·  70  ·  35.40  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  transferFrom                  ·  -  ·  -  ·  72141  ·  1  ·  11.97  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateEventState              ·  -  ·  -  ·  31032  ·  1  ·  5.15  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenState              ·  35439  ·  35450  ·  35445  ·  2  ·  5.88  │
····················|···························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenStateBatch         ·  -  ·  -  ·  3812565  ·  1  ·  632.50  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  burn                          ·  -  ·  -  ·  29089  ·  1  ·  4.83  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  buy                           ·  54250  ·  71350  ·  65884  ·  122  ·  10.93  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  increaseAllowance             ·  46523  ·  46547  ·  46546  ·  71  ·  7.72  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  mint                          ·  -  ·  -  ·  70470  ·  2  ·  11.69  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  sell                          ·  -  ·  -  ·  35368  ·  1  ·  5.87  │
····················|···························|·············|·············|·············|···············|··············
|  TIX                ·  updateRates                   ·  -  ·  -  ·  31841  ·  1  ·  5.28  │
····················|···························|·············|·············|·············|···············|··············
|  Deployments                                  ·                                         ·  % of limit   ·             │
················································|·············|·············|·············|···············|··············
|  ExchangeSignature                            ·  893597  ·  893609  ·  893603  ·  3 %  ·  148.25  │
················································|·············|·············|·············|···············|··············
|  ExchangeV1                                   ·  2179949  ·  2179973  ·  2179969  ·  7.3 %  ·  361.66  │
················································|·············|·············|·············|···············|··············
|  ExchangeV2                                   ·  -  ·  -  ·  1255857  ·  4.2 %  ·  208.35  │
················································|·············|·············|·············|···············|··············
|  Identity                                     ·  -  ·  -  ·  763699  ·  2.5 %  ·  126.70  │
················································|·············|·············|·············|···············|··············
|  IdentitySignature                            ·  -  ·  -  ·  718828  ·  2.4 %  ·  119.25  │
················································|·············|·············|·············|···············|··············
|  Ticketing                                    ·  3873364  ·  3873388  ·  3873386  ·  12.9 %  ·  642.59  │
················································|·············|·············|·············|···············|··············
|  TIX                                          ·  1447728  ·  1447740  ·  1447729  ·  4.8 %  ·  240.18  │
·-----------------------------------------------|-------------|-------------|-------------|---------------|-------------·
```



```
·-----------------------------------------------|---------------------------|-------------|------------------------------·
|             Solc version: 0.8.12              ·  Optimizer enabled: true  ·  Runs: 200  ·  Block limit: 30000000 gas  │
························································|···························|·············|······························
|  Methods                                      ·               30 gwei/gas               ·       1.36 chf/matic         │
······················|·························|·············|·············|·············|···············|··············
|  Contract           ·  Method                 ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithoutSignature  ·     -  ·          -  ·      81741  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithSignature    ·     -  ·          -  ·      88286  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptResale           ·          -  ·          -  ·     157746  ·            1  ·       0.01  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptSwap             ·          -  ·          -  ·      87081  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelResale           ·          -  ·          -  ·      30765  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelSwap             ·          -  ·          -  ·      30826  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createResale           ·      62815  ·      88440  ·      83772  ·           11  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createSwap             ·      65798  ·      91390  ·      73467  ·           10  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV2         ·  resell                 ·          -  ·          -  ·     186680  ·            1  ·       0.01  │
······················|·························|·············|·············|·············|···············|··············
|  ExchangeV2         ·  swap                   ·          -  ·          -  ·     111984  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Identity           ·  register               ·      54636  ·      54704  ·      54663  ·          247  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Identity           ·  revoke                 ·          -  ·          -  ·      30582  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Identity           ·  unregister             ·      27665  ·      27896  ·      27781  ·            2  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithoutProof   ·          -  ·          -  ·      46822  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithProof      ·          -  ·          -  ·      54515  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  approve                ·      48800  ·      48812  ·      48808  ·           10  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  burn                   ·          -  ·          -  ·      46184  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  burnBatch              ·          -  ·          -  ·    7243121  ·            1  ·       0.30  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  mint                   ·     173236  ·     178048  ·     173709  ·           52  ·       0.01  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  mintBatch              ·          -  ·          -  ·   28984724  ·            5  ·       1.18  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  registerEvent          ·     197563  ·     214663  ·     213381  ·           70  ·       0.01  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  transferFrom           ·          -  ·          -  ·      72141  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateEventState       ·          -  ·          -  ·      31032  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenState       ·      35439  ·      35450  ·      35445  ·            2  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenStateBatch  ·          -  ·          -  ·    3812565  ·            1  ·       0.16  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  burn                   ·          -  ·          -  ·      29089  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  buy                    ·      54250  ·      71350  ·      65884  ·          122  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  increaseAllowance      ·      46523  ·      46547  ·      46546  ·           71  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  mint                   ·          -  ·          -  ·      70470  ·            2  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  sell                   ·          -  ·          -  ·      35368  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  TIX                ·  updateRates            ·          -  ·          -  ·      31841  ·            1  ·       0.00  │
······················|·························|·············|·············|·············|···············|··············
|  Deployments                                  ·                                         ·  % of limit   ·             │
························································|·············|·············|·············|···············|··············
|  ExchangeSignature                            ·     893597  ·     893609  ·     893603  ·          3 %  ·       0.04  │
························································|·············|·············|·············|···············|··············
|  ExchangeV1                                   ·    2179949  ·    2179973  ·    2179969  ·        7.3 %  ·       0.09  │
························································|·············|·············|·············|···············|··············
|  ExchangeV2                                   ·          -  ·          -  ·    1255857  ·        4.2 %  ·       0.05  │
························································|·············|·············|·············|···············|··············
|  Identity                                     ·          -  ·          -  ·     763699  ·        2.5 %  ·       0.03  │
························································|·············|·············|·············|···············|··············
|  IdentitySignature                            ·          -  ·          -  ·     718828  ·        2.4 %  ·       0.03  │
························································|·············|·············|·············|···············|··············
|  Ticketing                                    ·    3873364  ·    3873388  ·    3873386  ·       12.9 %  ·       0.16  │
························································|·············|·············|·············|···············|··············
|  TIX                                          ·    1447728  ·    1447740  ·    1447729  ·        4.8 %  ·       0.06  │
·-----------------------------------------------|-------------|-------------|-------------|---------------|-------------·
```



```
·-----------------------------------------|---------------------------|-------------|-----------------------------·
|          Solc version: 0.8.12           ·  Optimizer enabled: true  ·  Runs: 200  ·  Block limit: 30000000 gas  │
··········································|···························|·············|······························
|  Methods                                ·         45 gwei/gas       ·           70.19 chf/avax                  │
····················|·····················|·············|·············|·············|···············|··············
|  Contract         ·  Method             ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithoutSignature  ·  -  ·  -  ·  81741  ·  1  ·  0.26  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithSignature     ·  -  ·  -  ·  88286  ·  1  ·  0.28  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  acceptResale       ·          -  ·          -  ·     157746  ·            1  ·       0.50  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  acceptSwap         ·          -  ·          -  ·      87081  ·            1  ·       0.28  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  cancelResale       ·          -  ·          -  ·      30765  ·            1  ·       0.10  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  cancelSwap         ·          -  ·          -  ·      30826  ·            1  ·       0.10  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  createResale       ·      62815  ·      88440  ·      83772  ·           11  ·       0.26  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1       ·  createSwap         ·      65798  ·      91390  ·      73467  ·           10  ·       0.23  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV2       ·  resell             ·          -  ·          -  ·     186680  ·            1  ·       0.59  │
····················|·····················|·············|·············|·············|···············|··············
|  ExchangeV2       ·  swap               ·          -  ·          -  ·     111984  ·            1  ·       0.35  │
····················|·····················|·············|·············|·············|···············|··············
|  Identity         ·  register           ·      54636  ·      54704  ·      54663  ·          247  ·       0.17  │
····················|·····················|·············|·············|·············|···············|··············
|  Identity         ·  revoke             ·          -  ·          -  ·      30582  ·            1  ·       0.10  │
····················|·····················|·············|·············|·············|···············|··············
|  Identity         ·  unregister         ·      27665  ·      27896  ·      27781  ·            2  ·       0.09  │
····················|·····················|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithoutProof  ·        -  ·          -  ·      46822  ·            1  ·       0.15  │
····················|·····················|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithProof     ·        -  ·          -  ·      54515  ·            1  ·       0.17  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  approve            ·      48800  ·      48812  ·      48808  ·           10  ·       0.15  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  burn               ·          -  ·          -  ·      46184  ·            1  ·       0.15  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  burnBatch          ·          -  ·          -  ·    7243121  ·            1  ·      22.88  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  mint               ·     173236  ·     178048  ·     173709  ·           52  ·       0.55  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  mintBatch          ·          -  ·          -  ·   28984724  ·            5  ·      91.55  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  registerEvent      ·     197563  ·     214663  ·     213381  ·           70  ·       0.67  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  transferFrom       ·          -  ·          -  ·      72141  ·            1  ·       0.23  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  updateEventState   ·          -  ·          -  ·      31032  ·            1  ·       0.10  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  updateTokenState   ·      35439  ·      35450  ·      35445  ·            2  ·       0.11  │
····················|·····················|·············|·············|·············|···············|··············
|  Ticketing        ·  updateTokenStateBatch  ·       -  ·          -  ·    3812565  ·            1  ·      12.04  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  burn               ·          -  ·          -  ·      29089  ·            1  ·       0.09  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  buy                ·      54250  ·      71350  ·      65884  ·          122  ·       0.21  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  increaseAllowance  ·      46523  ·      46547  ·      46546  ·           71  ·       0.15  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  mint               ·          -  ·          -  ·      70470  ·            2  ·       0.22  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  sell               ·          -  ·          -  ·      35368  ·            1  ·       0.11  │
····················|·····················|·············|·············|·············|···············|··············
|  TIX              ·  updateRates        ·          -  ·          -  ·      31841  ·            1  ·       0.10  │
····················|·····················|·············|·············|·············|···············|··············
|  Deployments                            ·                                         ·  % of limit   ·             │
··········································|·············|·············|·············|···············|··············
|  ExchangeSignature                      ·     893597  ·     893609  ·     893603  ·          3 %  ·       2.82  │
··········································|·············|·············|·············|···············|··············
|  ExchangeV1                             ·    2179949  ·    2179973  ·    2179969  ·        7.3 %  ·       6.89  │
··········································|·············|·············|·············|···············|··············
|  ExchangeV2                             ·          -  ·          -  ·    1255857  ·        4.2 %  ·       3.97  │
··········································|·············|·············|·············|···············|··············
|  Identity                               ·          -  ·          -  ·     763699  ·        2.5 %  ·       2.41  │
··········································|·············|·············|·············|···············|··············
|  IdentitySignature                      ·          -  ·          -  ·     718828  ·        2.4 %  ·       2.27  │
··········································|·············|·············|·············|···············|··············
|  Ticketing                              ·    3873364  ·    3873388  ·    3873386  ·       12.9 %  ·      12.23  │
··········································|·············|·············|·············|···············|··············
|  TIX                                    ·    1447728  ·    1447740  ·    1447729  ·        4.8 %  ·       4.57  │
·-----------------------------------------|-------------|-------------|-------------|---------------|-------------·
```



```
·-------------------------------------------|---------------------------|-------------|-----------------------------·
|           Solc version: 0.8.12            ·  Optimizer enabled: true  ·  Runs: 200  ·  Block limit: 30000000 gas  │
············································|···························|·············|······························
|  Methods                                  ·             5 gwei/gas                  ·       339.57 chf/bnb        │
······················|·····················|·············|·············|·············|···············|··············
|  Contract           ·  Method             ·  Min        ·  Max        ·  Avg        ·  # calls      ·  chf (avg)  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithoutSignature  ·  -  ·  -  ·  81741  ·  1  ·  0.14  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeSignature  ·  createResaleWithSignature  ·  -  ·  -  ·  88286  ·  1  ·  0.15  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptResale       ·          -  ·          -  ·     157746  ·            1  ·       0.27  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  acceptSwap         ·          -  ·          -  ·      87081  ·            1  ·       0.15  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelResale       ·          -  ·          -  ·      30765  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  cancelSwap         ·          -  ·          -  ·      30826  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createResale       ·      62815  ·      88440  ·      83772  ·           11  ·       0.14  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV1         ·  createSwap         ·      65798  ·      91390  ·      73467  ·           10  ·       0.12  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV2         ·  resell             ·          -  ·          -  ·     186680  ·            1  ·       0.32  │
······················|·····················|·············|·············|·············|···············|··············
|  ExchangeV2         ·  swap               ·          -  ·          -  ·     111984  ·            1  ·       0.19  │
······················|·····················|·············|·············|·············|···············|··············
|  Identity           ·  register           ·      54636  ·      54704  ·      54663  ·          247  ·       0.09  │
······················|·····················|·············|·············|·············|···············|··············
|  Identity           ·  revoke             ·          -  ·          -  ·      30582  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  Identity           ·  unregister         ·      27665  ·      27896  ·      27781  ·            2  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithoutProof  ·       -  ·          -  ·      46822  ·            1  ·       0.08  │
······················|·····················|·············|·············|·············|···············|··············
|  IdentitySignature  ·  registerWithProof  ·          -  ·          -  ·      54515  ·            1  ·       0.09  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  approve            ·      48800  ·      48812  ·      48808  ·           10  ·       0.08  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  burn               ·          -  ·          -  ·      46184  ·            1  ·       0.08  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  burnBatch          ·          -  ·          -  ·    7243121  ·            1  ·      12.30  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  mint               ·     173236  ·     178048  ·     173709  ·           52  ·       0.29  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  mintBatch          ·          -  ·          -  ·   28984724  ·            5  ·      49.21  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  registerEvent      ·     197563  ·     214663  ·     213381  ·           70  ·       0.36  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  transferFrom       ·          -  ·          -  ·      72141  ·            1  ·       0.12  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  updateEventState   ·          -  ·          -  ·      31032  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenState   ·      35439  ·      35450  ·      35445  ·            2  ·       0.06  │
······················|·····················|·············|·············|·············|···············|··············
|  Ticketing          ·  updateTokenStateBatch  ·      -  ·          -  ·    3812565  ·            1  ·       6.47  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  burn               ·          -  ·          -  ·      29089  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  buy                ·      54250  ·      71350  ·      65884  ·          122  ·       0.11  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  increaseAllowance  ·      46523  ·      46547  ·      46546  ·           71  ·       0.08  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  mint               ·          -  ·          -  ·      70470  ·            2  ·       0.12  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  sell               ·          -  ·          -  ·      35368  ·            1  ·       0.06  │
······················|·····················|·············|·············|·············|···············|··············
|  TIX                ·  updateRates        ·          -  ·          -  ·      31841  ·            1  ·       0.05  │
······················|·····················|·············|·············|·············|···············|··············
|  Deployments                              ·                                         ·  % of limit   ·             │
············································|·············|·············|·············|···············|··············
|  ExchangeSignature                        ·     893597  ·     893609  ·     893603  ·          3 %  ·       1.52  │
············································|·············|·············|·············|···············|··············
|  ExchangeV1                               ·    2179949  ·    2179973  ·    2179969  ·        7.3 %  ·       3.70  │
············································|·············|·············|·············|···············|··············
|  ExchangeV2                               ·          -  ·          -  ·    1255857  ·        4.2 %  ·       2.13  │
············································|·············|·············|·············|···············|··············
|  Identity                                 ·          -  ·          -  ·     763699  ·        2.5 %  ·       1.30  │
············································|·············|·············|·············|···············|··············
|  IdentitySignature                        ·          -  ·          -  ·     718828  ·        2.4 %  ·       1.22  │
············································|·············|·············|·············|···············|··············
|  Ticketing                                ·    3873364  ·    3873388  ·    3873386  ·       12.9 %  ·       6.58  │
············································|·············|·············|·············|···············|··············
|  TIX                                      ·    1447728  ·    1447740  ·    1447729  ·        4.8 %  ·       2.46  │
·-------------------------------------------|-------------|-------------|-------------|---------------|-------------·
```