Project Report

INF219 Project in Informatics I

A Blockchain-Based Ticket System

June 2020

Carl August Gjørsvik
University of Bergen

Joakim Algrøy
University of Bergen

Supervisor

Chunlei Li, Ph.D.
University of Bergen

Abstract

Ticket Sale

The ticket sale industry has always been subject to ticket theft, illegal resale and forgery. In today's market more than half of the population of Norway buy tickets on the internet[1] and in the USA, the market size of online ticket sales has passed 13.9 billion USD annually.[2] It is apparent that transaction processes need to be extremely precise and secure to avoid financial loss, either by the customer, the seller or both.

The ticket seller can be an independent party to the event host, and the ticket seller may use another independent service to process payment, which means there are often 2 layers which need to be trusted (and paid) between the customer and the host.

There is also the annoyance to customers that there are different systems where the customer must keep track of different accounts and payment options. And some claim that a certain international ticket seller entity is conducting unethical practices – such as obtaining a complete or near-monopoly on event locations in certain areas. When controlling venues, the company can then set the price of its services freely, without competition.[3]

Another issue is the resale of tickets between attendees. When buying tickets from someone other than the official seller you run the risk of the ticket already having been used when you show up at the event, or it could have been illegally forged. Selling your ticket to a stranger online can also be difficult, where the seller risks not getting paid if they send the ticket first, or the buyer risks not receiving the ticket if they send payment first.

Blockchain

A blockchain is a decentralized database or ledger, typically used to facilitate peer-to-peer financial transactions in the form of a cryptocurrency. Some cryptocurrency implementations have the ability to publish smart contracts on the blockchain, where a public and unchangeable contract can be created and is automatically enforced by the nodes in the

[1] "07001: Use of Internet for buying or ordering goods and ... - SSB." 30 Aug. 2018, https://www.ssb.no/en/statbank/table/07001. Accessed 21 May. 2020.
[2] "Online Event Ticket Sales in the US - Market Size | IBISWorld." https://www.ibisworld.com/industry-statistics/market-size/online-event-ticket-sales-united-states/. Accessed 21 May. 2020.
[3] "Ticketmaster, Live Nation Accused of Monopoly Practices in ...." 30 Apr. 2020, https://www.ticketnews.com/2020/04/ticketmaster-live-nation-accused-of-monopoly-practices-in-new-lawsuit/. Accessed 21 May. 2020.

network.[4] Blockchains are particularly suited for removing the need for trust in third parties. In our case, we want to remove the need to trust a third party with your money and data when buying tickets for an event.

The peer-to-peer nature of a cryptocurrency could allow an event organizer to sell tickets directly to guests, without relying on a third party. Our role will be to provide software to facilitate such transactions, without taking part in the financial transactions themselves. Creating a smart contract for an event would allow a transparent sale of tickets, where information like the start of the sale, the number of tickets, and the price can be publicly available. Users will be able to buy tickets from this smart contract by sending a cryptocurrency transaction to the contract address.

Ownership of a ticket would be tied to an address, which is a representation of the public key of a public/private key pair. Ownership of an address can be proven by signing a predefined message using the corresponding private key.

It also might be possible to facilitate resale of tickets through a smart contract. Someone wishing to sell a ticket could announce that with the help of the smart contract, and others looking to buy could buy that ticket if the original tickets are sold out. This removes the need for the buyer and seller to trust each other in the case of resale of tickets. One could also add restrictions on the price of resold tickets; however this could of course be avoided by selling each other the private keys directly instead of using the smart contract, unless the addresses are otherwise authenticated upon purchase.

1 Introduction

Smart Contracts

While blockchains were initially introduced for the purpose of peer-to-peer payments[5], Ethereum expanded the scope of this technology to include *smart contracts*, a term which was conceptualized by Nick Szabo in 1997.[6] This allows for information storage and transactions governed by programmed rules set in a contract. The correct execution of these rules are enforced by the consensus protocol just like it enforces secure payments.[7]

Ethereum smart contracts have a state consisting of its balance and an associated data storage on the blockchain. A contract can be called using its public address; if the call transfers funds or changes the state of the data storage, this interaction must be agreed upon by the network's nodes through the consensus protocol. State changes in contracts are

[4] "Ethereum Whitepaper | Ethereum.org." 27 May. 2020, https://ethereum.org/whitepaper/. Accessed 7 Jun. 2020.

[5] "Bitcoin: A Peer-to-Peer Electronic Cash System - Bitcoin.org." https://bitcoin.org/bitcoin.pdf. Accessed 22 May. 2020.

[6] "The Idea of Smart Contracts | Satoshi Nakamoto Institute." https://nakamotoinstitute.org/literature/the-idea-of-smart-contracts/. Accessed 22 May. 2020.

[7] "White Paper · ethereum/wiki Wiki · GitHub." 17 Jun. 2019, https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 22 May. 2020.

computed by the Ethereum Virtual Machine (EVM) on network nodes, using a low-level language referred to as EVM code. Developers can write contracts for Ethereum in high-level languages such as Solidity[8], which is the most popular of the languages with EVM-code compilers.

Project Specification

We defined the following overall goal and four phases to achieve said goal as our approach to the development of the project:

> **Goal**: Allow *hosts* to sell *tickets* on the blockchain for some *event*. Customers who have bought tickets may use the ownership of their wallet as proof of ownership of bought tickets (private key verification). Hosts should have a list of addresses which own tickets, or an equivalent way to verify customers. Ticket theft should be infeasible as one would have to get hold of the customer's private key.
>
> **Phase 1**: Develop a smart contract which has a set number of tickets and a price upon deployment. It should have a payable method where customers can attempt to pay for a ticket. The contract should return excessive amounts. A public call method is needed to check the availability of tickets. The creator should be able to call the contract to receive a list of addresses which have bought tickets.
>
> **Phase 2**: Advance the contract such that it can be used simultaneously for a number of different events. Anyone should be able to create events in the contract by calling a payable function to create events with the necessary arguments. Events must have a unique identifier.
>
> **Phase 3**: Create a web application layer for this service such that users can connect to and interact with the smart contract service using their Metamask wallet. Event hosts can create events through a simple interface and customers can buy tickets with the click of a button. This site can display the information which is on the blockchain about each event (e.g. title, date, number of available tickets).
>
> **Phase 4**: Expand the web app to include hosting of information which is not on the blockchain. Users can log in to the app and view more elaborate information of events etc. The customer app should be able to display some proof of ticket ownership which can be scanned - e.g. QR-code of signature of event ID, optionally event ID and a time stamp.

Phase four was defined in the plan, though it was apparent it was not achievable within the scope of this project, but rather an idea for further development of this project, which is a proof of concept. With an eventual production state on the live Ethereum network, the smart contract system could be utilized by anyone. While other parties such as travel agencies or concert hosts could find it useful to create their own web interfaces interacting with the

[8] "ethereum/solidity - GitHub." https://github.com/ethereum/solidity. Accessed 22 May. 2020.

system, while storing additional off-blockchain (e.g. conventional database) information tied to their *events*. An event host could for example want to display pictures related to an event in the browsing interface for customers, and it would be possible to store encoded pictures on the blockchain, but to keep the transaction fees to a minimum, it is preferable to have non-essential data stored off-blockchain.

Development Environment and Tools

For the development of smart contracts for the Ethereum network, Solidity is the obvious choice as it is a well established and documented language, created by contributors of the Ethereum network core, specifically to compile for the Ethereum Virtual Machine.

Truffle[9] is a smart contract lifecycle management tool, facilitating automated testing and deployment to blockchain. Automated tests are written in JavaScript and Solidity - both are useful as tests written in Solidity can perform sanity tests directly on the contract state and it's changes, while JavaScript tests can simulate external interaction with the contract using libraries such as Web3.js.[10]

The Truffle Suite also provides a tool named Ganache[11], which is used to run a local Ethereum test network. Truffle can deploy contracts to the locally run Ganache network to allow external tests, and tools such as MyEtherWallet[12] and MetaMask[13] can connect and interact with the contracts.

To deploy contracts to the public test network Ropsten, we created a deployment channel to an Infura Ethereum API.[14] Infura provides load-balanced access nodes to selected Ethereum networks including Ropsten and the Mainnet.

For the purpose of creating the web application layer, we have used React[15], a JavaScript library for building effective interactive user interfaces. The library Material-UI[16] provides a selection of customizable React components.

[9] "Truffle | Truffle Suite." https://www.trufflesuite.com/truffle. Accessed 23 May. 2020.
[10] "web3.js - Ethereum JavaScript API — web3.js 1.0.0 ...." https://web3js.readthedocs.io/. Accessed 23 May. 2020.
[11] "Ganache | Truffle Suite." https://www.trufflesuite.com/ganache. Accessed 23 May. 2020.
[12] "MyEtherWallet." https://www.myetherwallet.com/. Accessed 23 May. 2020.
[13] "MetaMask." https://metamask.io/. Accessed 23 May. 2020.
[14] "Infura." https://infura.io/. Accessed 23 May. 2020.
[15] "React." https://reactjs.org/. Accessed 3 Jun. 2020.
[16] "Material-UI." https://material-ui.com/. Accessed 3 Jun. 2020.

2 Design and Implementation

Project Architecture

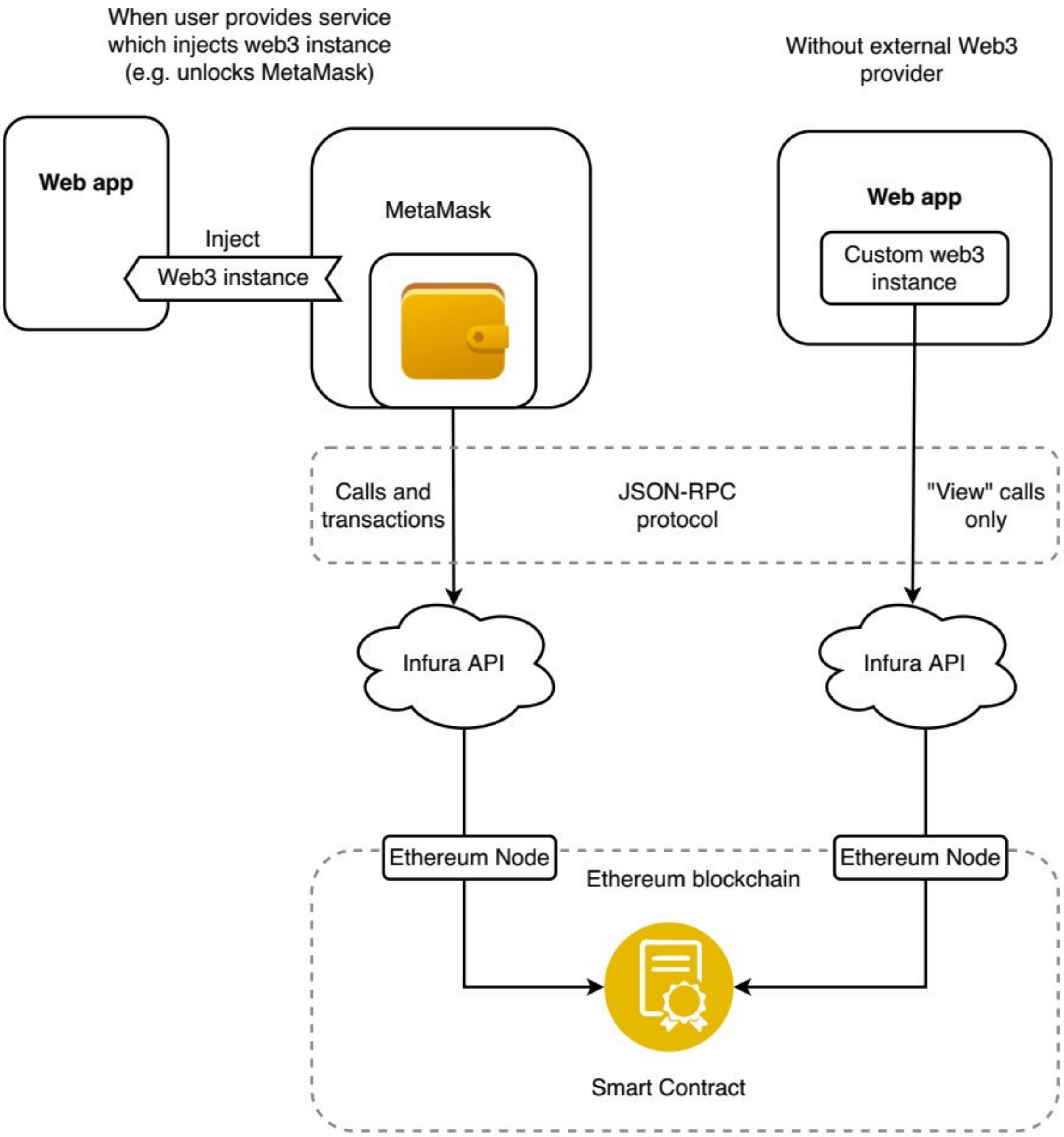

Figure 2.1: Communication architecture

The above illustration shows how the web application communicates with the smart contract deployed to the blockchain. It is separated into two “channels”, on the right is the default option, providing a view-only communication channel for users without a digital wallet or users who have not unlocked their wallet. The other option is enabled when the user connects a digital wallet providing an interactive channel.

Communication from either the custom Web3 instance or the one provided by a digital wallet, in this case MetaMask, is conducted in the form of a protocol called JSON-RPC[17] to a HTTPS API hosted by Infura. Infura relays the Remote Procedure Call (RPC) received through the JSON-RPC protocol to one of their Ethereum nodes, which broadcasts the transaction on the network.

Web Application

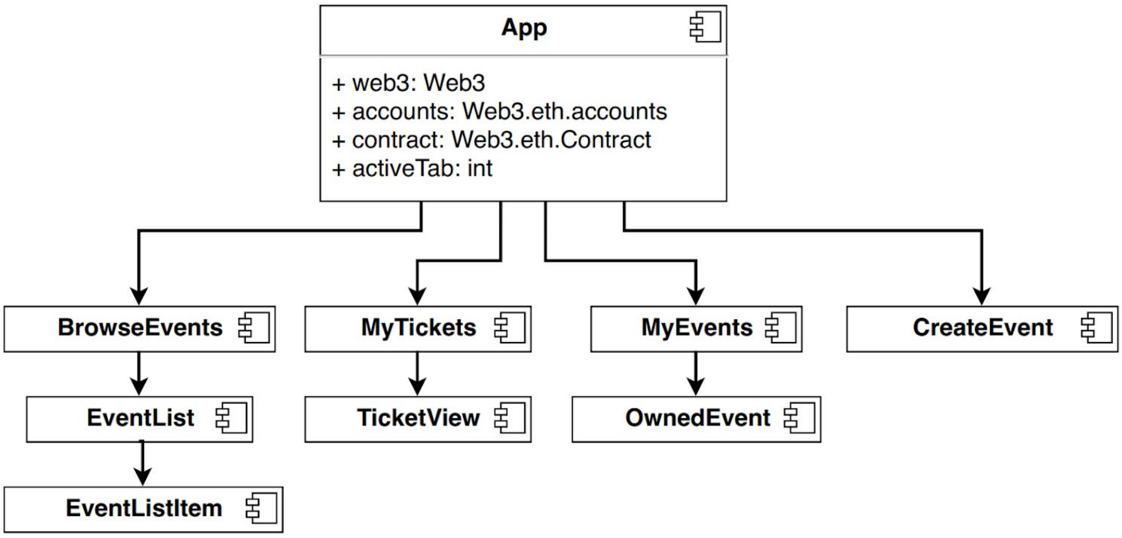

Figure 2.2: Simplified flow diagram of web application

The web application was created as a minimum viable product to serve as an interface for this project. It contains four main tabs which serve the following purposes:

1. Allow a user to browse all events stored in the contract, view basic information such as event title, deadline and tickets available, and buy tickets.

2. Display tickets owned by the user and provide the option to return tickets where possible.

[17] "JSON RPC · ethereum/wiki Wiki · GitHub." 24 Apr. 2020, https://github.com/ethereum/wiki/wiki/JSON-RPC?ref=hackernoon.com. Accessed 6 Jun. 2020.

3. Display events created by the user and provide options to control these events. These options include starting and stopping sale, withdrawing funds, adding tickets, changing ticket prices and delete events.

4. Allow a user to create events.

The application was created using React and Material-UI, where Material-UI provides a collection of premade customizable React components. The interface is made up of a *tab panel* where the user can choose one of four tabs described above, the content of which will be displayed below the tab panel. There is also an information banner at the top which will become visible if transactions are initiated, displaying transaction statuses; when transactions are pending, confirmed or failed. While the app is not connected to the user's MetaMask account, an option to connect is displayed, as well as a link to get MetaMask.

As shown in figure 2.2, *App* is the root component of the React component tree. All event data within the smart contract are extracted and buffered by the App component and distributed as *props*[18] to the necessary child components. The Web3 instance is also created and access is distributed similarly. An advantage of using React is that when the state of a component changes (e.g. an event is updated in the buffer or the web3 instance changes accounts on user input) the child components which receive that state as props will automatically have their props updated and re-render changes to the interface. As an example, if a new event is created in the contract, the app will notice the change and update its buffer, which will be distributed to child components and a new EventListItem will be created and displayed automatically.

[18] "Components and Props – React." https://reactjs.org/docs/components-and-props.html. Accessed 5 Jun. 2020.

Web3 and MetaMask

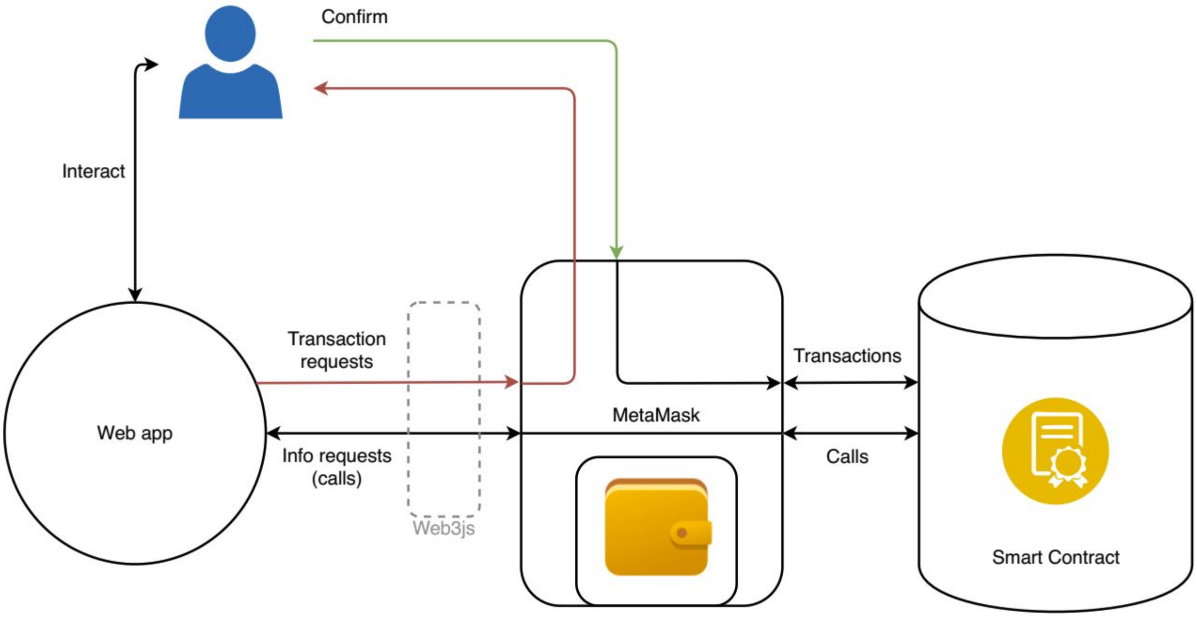

Figure 2.3: Sketch of user interaction flow

There are three main ways for a web application to connect to an Ethereum network node through the Web3 library; most modern web browsers provide an instance of Web3 with connection to the Ethereum Mainnet, MetaMask provides an instance of Web3 which connects the app to the network chosen in the MetaMask settings, and there is also the option to instantiate a custom Web3 connection with reference to a chosen network node. The browser-provided instance only allows for viewing information and not to call transactional functions as there are no accounts involved, and can be used to view event data in the contract on the Mainnet.

For the purpose of this project, we have used MetaMask to provide the main communication layer between the web application and the blockchain, although other alternatives exist (e.g. the Opera browser[19]). MetaMask functions as a digital wallet, handling the users keys in a manner which isolates sensitive data from the content of the browser[20].

To be able to interact with the system, a user must create a MetaMask account and import an existing Ethereum address key or generate a new address in the MetaMask wallet. When connecting MetaMask to the app, the user will be prompted to unlock an account for communication with the app - this unlocking simply provides the app with non-sensitive information related to the user's account such as public address and balance. After

[19] "Opera launches Reborn 3: the first desktop browser ready for ...." 9 Apr. 2019, https://press.opera.com/2019/04/09/opera-reborn-3-first-desktop-browser-ready-for-web-3-the-internet-of-the-future/. Accessed 29 May. 2020.
[20] "MetaMask Docs: Introduction." 28 Apr. 2020, https://docs.metamask.io/. Accessed 29 May. 2020.

unlocking, a user can initiate a transaction by for example clicking a button to buy tickets, then the app will “ask” MetaMask to perform the transaction which will trigger MetaMask to prompt the user with a choice to either confirm or reject the transaction.

While accessing the app without MetaMask installed or without unlocking any accounts, we have implemented a viewing solution to allow users to browse information related to stored events, i.e. users cannot create events, buy tickets or otherwise interact, but still see available events. This solution uses the third Web3 option described above - a custom instantiation of Web3 using an Infura API to access the Ropsten network for the time being. For deployment to the Mainnet this could either be changed to a Mainnet Infura API, or optionally use a browser-provided Web3 instance.

Mobile

The web application was created such that the same interface as shown in a browser adapts to mobile viewing. While most PC web browsers need an extension such as MetaMask to interact with the system, which is not available for the corresponding mobile web browsers, mobile users can access the system through for example the MetaMask app for Android and iOS.

Smart Contract

The main result of this project is a single smart contract intended to be published on the Ethereum blockchain. This contract provides data storage, a series of functions that can be called by sending a transaction to the contract’s Ethereum address, and some view functions (read only) that can be called without a transaction. We have split the functions into four sections; event host functions, customer functions, view functions, and internal functions.

Event Host Functions

Event host functions are the functions called by event hosts to create, modify, and delete events. To ensure only the owner of an event can modify or delete that event, we use the sender address of a transaction as authentication. We do this by using Solidity modifiers and the *require* statement. Modifiers can be applied to functions to perform some check or action before and/or after the body of the function is executed. The following is our definition of the onlyHost modifier, used to ensure only the host of an event can call a specific function using the corresponding event id.

```
modifier onlyHost(bytes32 event_id){
  require(events[event_id].owner == msg.sender,
    "Sender is not the owner of this event");
  _;
}
```

The underscore in the modifier definition sets when the code of the function the modifier is applied to should be executed. In this case a *require* statement is executed before anything else in the function, checking that the address calling the transaction is also the owner of the event with the provided event id. In Solidity, *require* is an assertion that throws an error when false, with the provided error message. If an error is thrown, any state modifications made in the function will be reverted.[21] As we will discuss in the security considerations, it is still good practice to have *require* statements at the top of a function, before any state is changed.

The create event function is a publicly available function to create a new event. This function does not cost anything beyond the gas costs of executing the function itself. We allow the caller to choose an event id themselves in a parameter to the function, which is 32 bytes of arbitrary data. A frontend application for interacting with the contract can either choose to generate a random id, or allow the user to choose. We choose to let the frontend handle the choice of event ids as that provides slightly more flexibility, but we could have just as easily chosen to have an incrementing integer as an id. 32 bytes provide 2^{256} unique ids, which should be more than enough for the lifetime of the contract. Further, the create event function takes parameters such as an event title, the number of available tickets per ticket type, the ticket prices, the limit on tickets per customer, the deadline, as well as a few boolean parameters like whether the sale of tickets is activated, whether the per customer limit should be enforced, and whether ticket returns are active. After some checks on the validity of the parameter values, the new event is added to storage.

We provide four functions for modifying the properties of the event; stop sale, continue sale, add tickets, and change ticket price. This could easily be expanded to include others such as changing the deadline as well as disabling and enabling ticket returns. Though there is a concern that allowing the event host to disable ticket returns after the sale has already started might increase the trust customers need to have in the hosts. We feel a good compromise is to allow the event host to choose whether tickets can be returned when creating an event, but not allow them to change that decision mid sale.

The withdraw function allows an event host to withdraw the funds collected in the contract by ticket sale related to the specified event. This function is restricted, as all host functions, by a modifier such that only the host and no other addresses can call the function. It is also currently restricted to reject transactions until the deadline of the event has passed, which we implemented as a temporary solution, considering that it would be more ideal to have this be an option on creation of events. In that case, a host could create an event with the option to withdraw funds while selling tickets, before the deadline - this would naturally force the buyback option to be disabled as there potentially wouldn't be funds in the contract to buy back tickets.

Hosts can also call a function to delete events they have created. If the event has associated funds stored in the contract or less than a week has passed since the deadline, this function will fail. The choice of a week after deadline as time restriction on the deletion of events is a placeholder for further development. We argue that this time limit should eventually be an

[21] "Expressions and Control Structures - Solidity - Read the Docs." https://solidity.readthedocs.io/en/v0.6.8/control-structures.html. Accessed 19 May. 2020.

option for hosts to choose, but likely with some minimum, for the information to remain visible and transparent on the blockchain for some time after an event. There is currently no real incentive for hosts to delete their events. We discuss the implications of this in the evaluation section of this report.

Customer Functions

Customer functions are the functions called by customers of an event. There are only two such functions, the function for buying tickets and the function for returning tickets. These also make use of *require* statements, for example to make sure the ticket sale deadline has not passed. When calling the buy tickets function, the caller must send a transaction to the contract address with an amount of ether greater than or equal to the ticket price, in addition to the gas costs of the transaction. If the sender sends more ether than necessary, the excess amounts will be returned to the sender. Note that this all happens in a single ethereum transaction which the sender pays for. Similarly, the return tickets function will delete the sender as a customer for an event, add to the number of available tickets for the event, and refund the sender the price of the tickets. This is only possible if the buyback option has been enabled by the event host.

View Functions

View functions are not called through transactions on the ethereum blockchain, but are simply methods for reading data from the contract. These can, as in our project, for example be called from a web frontend in order to present data stored in the contract. In addition to the view functions we explicitly define, there are implicit view functions for each of the class variables. For example there is an implicit view function called event_id_list() which will return the value of the event_id_list variable.

We have a get_events function that returns a list of event ids for all the events in the contract. And a get_event_info function that, given an event id, returns more information about that event like the title, owner, deadline, available tickets etc. We also have a get_participation function, that returns a list of event ids that the caller owns tickets to.

The contract includes three view functions aimed at event hosts. View funds, which returns the current amount of funds tied to the event from ticket sales, get tickets, which takes an address as a parameter and returns the number of tickets held by that customer address, and get customers, which returns a list of addresses that own tickets to the event. All of these take the event id as a parameter.

While these functions are intended to be called by event hosts, they are simply returning publicly available data on the blockchain, so restricting them to just event hosts using a modifier does not provide any actual secrecy. To demonstrate this, we can look at our view function view_funds which currently has an onlyHost modifier, meaning only the host of the event in question can call that function. The following function call will throw an exception:

```
EventContract.view_funds(web3.util.toHex(“test_id”),
    { from: accounts[1] });
```

This calls view_funds with “test_id” as the parameter. The “from: accounts[1]” statement defines who is calling the function, and in this case “accounts[1]” is not the host, so an exception is thrown. However, using the implicit view functions of the class variables we can still get this information with the following call:

```
EventContract.events(web3.utils.toHex("test_id"),
    { from: accounts[1] });
```

Which returns:

```
Result {
  owner: '0xbC9D3aB4Fd77eD581A7af14990A57739eB327204',
  event_id: '0x74657374',
  title: '0x54657374',
  index: <BN: 0>,
  max_per_customer: <BN: 1>,
  funds: <BN: 0>,
  exists: true,
  sale_active: true,
  buyback_active: true,
  per_customer_limit: false,
  deadline: <BN: 5ecfa748>
}
```

Here, we can see that the funds in the event are zero. This shows how the view functions are simply a convenience, and should not be relied on to limit access to data. The blockchain data is publicly available and can always be retrieved through other means if a view function fails due to a requirement on who is calling the function.

Internal Functions

These are only possible to call from within other functions in the same contract. They are simply used to avoid duplicating code and to improve readability.

Data Storage

Our contract needs to store data on the blockchain. The primary data types we need to store are Events and Customers. Events have properties such as an event name, the number of available tickets, a deadline, etc. A Customer is someone who owns tickets to a specific event, and the Customer data type will have an ethereum address and an associated number of tickets. The Event data structure needs to have a list of Customers who own tickets to that event.

In a typical software solution this data might be stored in a relational database, however this data has to be stored on the blockchain, and Solidity does not provide a database abstraction. Blockchain data in Solidity is stored as class variables. Solidity provides the ability to define your own data structures with structs, and it supports arrays and mappings. We need to create a set of class variables that will allow us to efficiently create, read, update and delete Events and Customers. As we will touch on more in this report, efficiency is key due to the literal costs involved with every instruction in a called function. Our storage of Events and Customers is largely based on a pattern described by Rob Hitchens in a Medium blog post.[22] This pattern allows efficient manipulation of blockchain data. Here is a simplified version of the contract storage in Solidity:

```
contract EventContract {
  mapping(bytes32 => Event) events;
  bytes32[] event_id_list;

  struct Event {
    address payable owner;
    bytes32 event_id;
    uint index;
    address[] customers;
    mapping(address => Customer) tickets;
    (...)
  }

  struct Customer {
    address addr;
    uint index;
    uint64 num_tickets;
    (...)
  }
```

As shown, we store both a list of event ids as well as a mapping from event ids to the Event data structure. In Solidity, every possible key of a mapping can be read even if it has not been initialized. This means it is not possible to enumerate the existing keys, and we have to keep track of the event ids in an array if we wish to for example have a method returning a list of all existing events. To avoid having both a mapping and an array, we could just store a single array of Event structs, but that would drastically increase gas costs for any function that updates data in an event, as we would have to loop through the list to find the event in question. The mapping/key list solution does present a problem when deleting events, as we need to delete the event id from the key list without having to loop through the array. The solution to this is to store the index of the event id in the key list as a property of the Event struct. That way, we can swap the event id we are deleting with the last event id in the array, and decrease the length of the array by one.

[22] "Solidity CRUD- Part 1 - Rob Hitchens - Medium." https://medium.com/robhitchens/solidity-crud-part-1-824ffa69509a. Accessed 12 May. 2020.

The same pattern is repeated when storing Customers inside Events. This allows all the following functions to execute in constant time, minimizing gas costs: Creating events, modifying event data such as the ticket price, deleting events, buying tickets and returning tickets.

Security Considerations

Reentrancy Attacks

This is a vulnerability unique to smart contracts where an attacker can exploit badly structured contract functions with conditional transactions back to the caller. This type of attack also makes use of the fact that a transfer call to an address executes the *fallback* function[23] if the address is a contract. The fallback function is executed when a contract address is called without naming a specific function. In Solidity, the fallback function is defined by implementing a function with no name, of which the compiler accepts only one.

The following is an example to show how a reentrancy attack works.

A simplified “bank” contract vulnerable to reentrancy attacks:

```
contract ExampleBank {
    mapping (address => uint) public balance;
    function deposit() { balance[msg.sender] += msg.amount }
    function withdraw() {
        msg.sender.call.value(balance[msg.sender]);
        Balance[msg.sender] = 0;
    }
}
```

A simplified contract constructed to exploit the ExampleBank contract:

```
contract RobBank {
    ExampleBank public bank = ExampleBank(...address...);
    function deposit() { bank.deposit.call.value(msg.amount); }
    function() { bank.withdraw.call(); }
}
```

The root of the vulnerability in the ExampleBank contract is that the balance mapping is updated after the actual withdrawal transaction. To execute this attack, the attacker must first deposit any amount through the RobBank contract, to register it with a balance in the “bank”. Then the attack can be started by calling the fallback function of the RobBank contract. The malicious contract will call the bank’s withdrawal function which in turn

[23] "Contracts — Solidity 0.5.12 documentation - Read the Docs." https://solidity.readthedocs.io/en/v0.5.12/contracts.html. Accessed 22 May. 2020.

executes a transaction back to the malicious contract's fallback function creating a loop of withdrawals before the bank contract can reach the code to update the relevant balance, potentially withdrawing the entire bank contract's balance. The attacker's contract would also need additional code to be able to withdraw the stolen balance, but this was omitted for simplicity.

An example of this type of attack is *the DAO attack*, where an equivalent of approximately $60 million was stolen[24], after which the community of the Ethereum network created a hard fork, resulting in the Ethereum (ETH) network invalidating the theft transactions and a new network emerging; Ethereum Classic (ETC)[25]. The ETC blockchain did not alter its history and the attack remained successful on the fork.

Integer Overflow Exploitation

Solidity has signed and unsigned integer types with sizes ranging from 8 bits to 256 bits.[26] These are susceptible to integer overflows and underflows, meaning arithmetic expressions can return incorrect results if the actual result is outside of the range that can be represented by the type. Consider the following snippet from an early version of our contract, in the function for buying tickets:

```
function buy_tickets(uint64 requested_num_tickets)
external payable {
    require(msg.value >=
    requested_num_tickets*ticket_price,
    "Not enough ether was sent.");
```

Here, msg.value is the amount of ether sent, and requested_num_tickets is a parameter chosen by the buyer. Both requested_num_tickets and ticket_price are unsigned 64-bit integers, thus, if the result of the multiplication in the require statement exceeds $2^{64}-1$ it will overflow and potentially allow the statement to be true even if the user has not sent enough ether. This effectively allows an attacker to steal tickets under the right circumstances. Assuming ticket_price is 2^{58} (not unreasonable as the unit is 10^{-18} ether), an attacker could set requested_num_tickets to 2^{6} or 64 and because $2^{58}\times 2^{6} = 2^{64}$, the result will overflow to zero, and the attacker will be allowed to purchase 64 tickets for free.

[24] "A survey of attacks on Ethereum smart contracts - Cryptology ...." https://eprint.iacr.org/2016/1007.pdf. Accessed 22 May. 2020.
[25] "The DAO, The Hack, The Soft Fork and The Hard Fork ...." 12 Mar. 2019, https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/. Accessed 22 May. 2020.
[26] "Types — Solidity 0.6.7 documentation - Read the Docs." https://solidity.readthedocs.io/en/v0.6.7/types.html. Accessed 25 May. 2020.

In this case we solved the problem by casting each of the integers to unsigned 128-bit integers before doing the multiplication:

```
function buy_tickets(uint64 requested_num_tickets) external
payable {
    require(msg.value >=
    uint128(requested_num_tickets)*uint128(ticket_price),
    "Not enough ether was sent.");
```

This way it is not possible for the expression to overflow, because the multiple of the max values of two 64-bit integers can still be expressed with 128 bits: $(2^{64} - 1)^2 < 2^{128}$. If our contract was more complex and manual analysis of each function was more difficult, we could employ a library like OpenZeppelin's SafeMath. SafeMath provides functions to replace arithmetic operators in Solidity, and will throw an exception that causes a revert when an integer overflow is encountered.[27] This does however come with some minor computational overhead that will increase gas costs. Simply relying on an exception will of course also cause potential bugs. In the example given above the overflow can still happen when the user has given perfectly valid parameters to the function, so throwing an exception would not be an acceptable solution in that case.

3 Evaluation

Security

Inherent Security of Ethereum

When performing transactions on the Ethereum blockchain, be it for purchases, gifts or other purposes, no sensitive information related to the digital wallet is transferred. On the other hand a more common, classical way of transferring funds on the internet is the use of credit cards - where sensitive information is often transferred and stored server-side. A consequence of the wide use of this payment method is that credit card fraud is quite common[28]. While criminals can steal stored credit card information or trick victims into giving up said information through malicious web sites, smart contracts do not store sensitive information related to customers and intercepted transactions do not hold information that can lead to theft.

Of course, a malicious developer may set up a *phishing* site, tricking victims into unlocking their digital wallet and paying for some non-existant or fraudulent service or item - but the criminal can still only get the amount the victim actively approves for transfer. If the case of

[27] "Math - OpenZeppelin Docs." https://docs.openzeppelin.com/contracts/3.x/api/math. Accessed 27 May. 2020.

[28] "Credit Card Fraud Statistics - Shift Processing." https://shiftprocessing.com/credit-card-fraud-statistics/. Accessed 3 Jun. 2020.

this phishing attack used credit card data, one fraudulent purchase would result in the attacker having control to spend the entire funds of the victim's account.

With regard to ticket forgery, the Ethereum system also provides protection; a contract's state and storage on the blockchain is immutable until a new block is agreed upon. The blockchain algorithm provides cryptographic proof that the new state and storage of a contract represented in a new block can only be derived by applying a change to the previous state while following the rules set in the contract. The only known way to forge or modify transactions is the *51% Attack*[29], in which an attacker must control a majority of all nodes operating the network (this is widely considered practically impossible on the Ethereum network). If an attacker were able to perform a 51% attack against Ethereum, the contract would no longer be protected against ticket theft and forgery, though we do not consider this attack vector an actual threat due to its “difficulty”.

Digital Wallets

With the use of digital wallets come some security advantages, while there are still risks. As discussed, sensitive information need not be exposed to perform payments as is the case with credit card transactions. Keeping the sensitive information - in this case the private key(s) - is the responsibility of the user, and most digital wallet hardware and software provide great tools and guides to help. For example, when creating a new account with the MetaMask browser extension, it will guide the user to create an account from a secure mnemonic seed, check that the user has stored the seed (to avoid loss of wallet control) and require the creation of a password to access the account. Most digital wallet providers will inform users of the importance of keeping the mnemonic seed secret and suggest the use of secure passwords. So the security can still be compromised by careless users, in the case of insecure passwords, insecure storage of mnemonics or users can be tricked into sharing their passwords or mnemonics.

Ticket Resale

Event hosts may want to disallow the resale of purchased tickets for several reasons; it could be to avoid third party resellers, potentially using bots to get hold of as many tickets as possible, to effectively increase the price of the event tickets for their own financial gain[30]. Another motivation may be to set prices below what the market demand would suggest, while limiting ticket purchases per person, to ensure that less financially fortunate customers can afford tickets.

When selling tickets on the ethereum blockchain, it is trivial to enforce these rules on a per-address basis - that is, to enforce a maximum number of tickets per address and to disable ticket resale to other addresses. Though the problems are still not solved. A user with the intent to buy tickets exceeding the limit allowed per customer, to later resell at an increased price, can simply generate a number of addresses and purchase the maximum

[29] "What Is a 51% Attack? | Binance Academy." https://academy.binance.com/security/what-is-a-51-percent-attack. Accessed 6 Jun. 2020.

[30] "Ticket Bots: Everything You Need to Know - Queue-it." https://queue-it.com/blog/ticket-bots/. Accessed 3 Jun. 2020.

amount of tickets from each address. Then the reseller can sell the private keys to these addresses at a chosen price, effectively reselling tickets for profit, circumventing the rules set in the smart contract.

As ethereum wallets can change hands, a system internal to the blockchain cannot enforce rules such as “maximum tickets per customer” - it can only enforce limits per address. We consider that in this regard, ticket sales on a blockchain face the same challenges as the traditional market, where user accounts registered with some level of identity authentication is needed to combat unwanted transfer of tickets between customers.

Fees

Making it cheap to use the contract was an important consideration during development as we wanted to make our system a desirable alternative to other ticket sale platforms, where fees as high as 20-30% are common.[31] As we will show in this section, a typical fee for buying a ticket through our contract would be around 1 euro. This is a constant fee, and not dependent on the price of the ticket, meaning if as an example tickets ranging from €10 to €100 were sold, fees would range from 1% to 10%.

The monetary value of fees when interacting with the contract depends on three factors; the gas needed to execute the called function, the gas price chosen by the caller, and the current price of ether. Gas is a unit used to measure computational work in the EVM, and each instruction in ethereum bytecode has a certain gas cost associated with it.[32] When calling a smart contract through an ethereum transaction, the sender must specify a *gas limit* and a *gas price*. The gas limit is a limit to how much computation the sender is willing to pay for. When a miner is processing a smart contract function call, it will continue to execute until either the function terminates or the gas used exceeds the specified gas limit. If the gas limit is exceeded, the function call will fail. The *gas price* specifies how much ether the sender wants to pay for each unit of gas, usually denoted in Gwei. The amount of ether needed to pay for a transaction is the gas used multiplied by the specified gas price. If the specified gas price is too low, miners might not process the transaction at all. Both gas limits and gas price are typically chosen automatically by digital wallets, including the MetaMask wallet. With both gas price and ether price fluctuating, the real world value of the transaction fees can vary significantly over longer periods.[33]

Given this fee structure, our goal when writing the contract is to minimize the gas used by the functions in the contract. This is achieved by minimizing the number of instructions executed, which can be achieved for example by eliminating loops. As mentioned above, different EVM bytecode instructions have different costs associated with them. Among the

[31] "U.S. GAO - Event Ticket Sales: Market Characteristics and ...." 14 May. 2018, https://www.gao.gov/products/GAO-18-347. Accessed 3 Jun. 2020.
[32] "Introduction to Smart Contracts - Solidity - Read the Docs." https://solidity.readthedocs.io/en/v0.6.8/introduction-to-smart-contracts.html. Accessed 3 Jun. 2020.
[33] "Ethereum Charts and Statistics | Etherscan." https://etherscan.io/charts. Accessed 5 Jun. 2020.

more expensive instructions are writing to contract storage.[34] Gas costs for some instructions are not constant, but based on formulas, and correlating Solidity code to EVM instructions is not very intuitive in the development process. Because of this, we decided to go with a more empirical approach to limit gas costs, by writing tests that measured the gas used by each function. Figure 3.1 shows the result of our gas cost test script with an initially empty contract. For create_ticket, buy_ticket and return_ticket an average gas cost over 100 function calls is used as the gas used can vary slightly.

Function	**Gas used**	**Transaction cost (ETH)***	**Transaction cost (EUR)****
create_event	300 861	Ξ0.00572	€1.23
buy_ticket	241 776	Ξ0.00459	€0.99
stop_sale	28 398	Ξ0.00054	€0.12
continue_sale	28 486	Ξ0.00054	€0.12
add_tickets	32 139	Ξ0.00061	€0.13
return_tickets	71 151	Ξ0.00135	€0.29
withdraw_funds	26 984	Ξ0.00051	€0.11
delete_event	70 067	Ξ0.00133	€0.29

Figure 3.1: Gas costs for contract functions

* Using the recommended standard gas price of 19 Gwei as of June 2nd 2020[35]

** Using an average price of ether across markets of £215.33 as of June 5th 2020[36]

The gas costs shown in this test are in our opinion very reasonable. An important point however is that this test was run with a brand new contract with no previous stored data. To demonstrate how the gas costs grow as the contract storage grows, we ran the same test after first creating 10 000 events, and have 100 accounts buy 100 tickets each to random events. The results are shown in figure 3.2. The “Gas increase” column shows the percentage increase in gas costs compared to calling the same function on an empty contract.

[34] "djrtwo/evm-opcode-gas-costs: Gas Costs from ... - GitHub." https://github.com/djrtwo/evm-opcode-gas-costs. Accessed 3 Jun. 2020.

[35] "ETH Gas Station | Consumer oriented metrics for the ...." https://ethgasstation.info/. Accessed 2 Jun. 2020.

[36] "Ethereum (ETH) price, charts, market cap, and other metrics ...." https://coinmarketcap.com/currencies/ethereum/markets/. Accessed 5 Jun. 2020.

Function	**Gas used**	**Gas increase**	**Transaction cost (ETH)***	**Transaction cost (EUR)****
create_event	300 936	0.02%	Ξ0.00572	€1.23
buy_ticket	321 193	32.85%	Ξ0.00610	€1.31
stop_sale	28 398	0%	Ξ0.00054	€0.12
continue_sale	28 486	0%	Ξ0.00054	€0.12
add_tickets	32 139	0%	Ξ0.00061	€0.13
return_tickets	106 704	49.97%	Ξ0.00203	€0.44
withdraw_funds	26 984	0%	Ξ0.00051	€0.11
delete_event	70 067	0%	Ξ0.00133	€0.29

Figure 3.2: Gas costs after simulating contract use

* Using the recommended standard gas price of 19 Gwei as of June 2nd 2020[37]

** Using an average price of ether across markets of £215.33 as of June 5th 2020[38]

While the gas costs remain manageable after the simulation, the percentage increase in gas used by the buy_ticket and return_tickets functions are a concern. The reason for the increase is that the contract keeps track of which events a given user owns tickets to. For simplicity's sake we implemented this simply by storing an array for each customer address with the event ids. This means when returning tickets the corresponding event id needs to be deleted from the array, and when buying tickets we need to check if the array already has that event id before adding it. We currently do both of these tasks with a for loop, causing an increase in gas costs when an address owns tickets to many separate events. While we consider it relatively unlikely that one address would own tickets to hundreds of events, it would still be worthwhile to fix this using a similar storage model to the one used for storing the event and customer structs.

Other than that we are very satisfied with the virtually constant cost of the create_event function, and the literally constant cost of the rest of the functions. These functions generally do relatively simple writes to storage. Due to our contract storage being structured to facilitate constant time writes, updates and deletions, the cost of these operations do not change even though the amount of data stored increases.

[37] "ETH Gas Station | Consumer oriented metrics for the ...." https://ethgasstation.info/. Accessed 2 Jun. 2020.

[38] "Ethereum (ETH) price, charts, market cap, and other metrics ...." https://coinmarketcap.com/currencies/ethereum/markets/. Accessed 5 Jun. 2020.

Contract Evolution

To mitigate an ever-expanding use of storage and to free up unique event IDs for reuse, the use of the *delete event* function is necessary, but unlikely to be called by event hosts due to the gas cost of its execution, lack of incentive to do so or other reasons. We consider two possible solutions to provide incentive for hosts to delete their events after they have taken place.

The first is to add a small proportional fee to creation of events and/or ticket purchases to build a profit in the contract, which can later be used to cover gas cost and possibly reward for anyone who calls a function with the purpose of deleting old events.

Another option is a sort of deposit upon creation of events. When a host creates an event through the contract, an amount of ETH must be deposited in addition to the gas cost - the deposit will be returned to the host when the function is called to delete the event, serving as incentive to delete old events.

Ticket Validation

When a customer arrives at an event location and is required to show their ticket to the host, the host should be able to verify the validity of that ticket. As previously explained in the project specification, the implementation of software to validate tickets could not be achieved in the scope of this project, but we have a clear idea of how it could be implemented.

A customer can prove that they own a given address by cryptographically signing a message using the corresponding private key.[39] The message might for example be the customers address and the event ID. An event host can then verify the signature, and use the *view* functions of the contract to check how many tickets the signed address owns. This process would be fast enough to perform in real-time while customers are verified for access to an event because the process only requires a *view* function and does not need to wait for transaction confirmation. The need for communication with the Ethereum network can even be eliminated if the host downloads the list of customers and their tickets from the contract, and stores it locally. The host can now be certain that the customer has paid for the ticket(s). Practically speaking, this could be implemented with a smartphone app which displays the signed message in the form of a QR-code that the host scans with their phone for their app to verify the signature.

[39] "Signing and Verifying Ethereum Signatures – Yos Riady." 16 Nov. 2018, https://yos.io/2018/11/16/ethereum-signatures/. Accessed 7 Jun. 2020.

4 Conclusion

It seems this system is a viable option for the sale of tickets online. Compared to the current state of the online ticket sale industry, it should bring more transactional security, and better proof against forgery and theft. On the other hand, the problem of resale must still be addressed externally by some form of identity authentication.

The solution could also remove the need for layers of services between hosts and customers, which should reduce operating costs, and increase transparency with regard to fees and actual ticket cost.

The missing piece of the puzzle considering smart contract applications' validity as contenders to current industries might simply be the lack of adoption and accessibility. It still requires some technical know-how to be able to create and use a digital wallet, and to exchange other currencies to get ETH. This lack of adoption is apparent when looking at certain statistics; while the current number of active Visa credit cards worldwide is 1.142 billion[40], the current number of active Ethereum addresses is in the range of 400 to 600 thousand[41] and there are only 100 million unique Ethereum addresses ever activated.[42]

Future Development

Ticket resale

An interesting possibility for expanding the functionality of the contract is to create an on-chain market for ticket resale between customers. If the event host has not enabled ticket returns, customers might want to sell their ticket to someone else, either at the original price or at a discount. As discussed in the abstract, selling tickets online involves a lot of trust in the other party, but doing so through a smart contract could be a completely trustless transaction where the seller is guaranteed to receive payment if the ticket is sold, and the buyer is guaranteed to receive a valid ticket if the payment goes through.

There are several possible ways to implement this. One could for example implement a queuing system, where customers who were not able to get a ticket before they sold out, could pay to join a queue, and customers who no longer wanted their ticket could sell them to the person in front of the queue. Another method would be to implement a stock-market like orderbook, where buyers could put in orders for tickets at a certain price, and sellers could either accept those orders, or put in their own sell orders at a higher price. Either way,

[40] "Visa Inc. Q2 2020 Operational Performance Data." https://s1.q4cdn.com/050606653/files/doc_financials/2020/q2/Visa-Inc.-Q2-2020-Operational-Performance-Data.pdf. Accessed 5 Jun. 2020.

[41] "Ethereum Active Addresses chart - BitInfoCharts." https://bitinfocharts.com/comparison/ethereum-activeaddresses.html. Accessed 5 Jun. 2020.

[42] "Ethereum Unique Addresses Chart | Etherscan." https://etherscan.io/chart/address. Accessed 5 Jun. 2020.

it would be beneficial to limit the resale price to the original value, to avoid people buying as many tickets as possible only to sell them at a higher price. While implementing such a system does not prevent people from selling tickets at a higher price through other channels, it at least provides regular customers the ability to sell their ticket safely if they no longer need it.

Separating storage and logic

The code of a smart contract cannot be changed when it's been deployed on the blockchain. An issue we have not addressed in this project is what we would do if we found a bug or wanted to add a new feature to the contract. A potential improvement on the current situation would be to separate out storage to a separate contract from the logic. That way, a change in logic could happen without having to migrate all existing data to a new contract. This does come with a new issue of keeping track of contract versions, but systems for handling this has been developed in the past.

The "Five Types Model" is a proposed model for allowing upgrades and extensions of existing smart contracts.[43] The essential idea is to split smart contracts in a system into five different types: Database contracts, Controller contracts, Contract managing contracts (CMCs), Application logic contracts (ALCs), and Utility contracts. Interaction with the database would only happen from the controller contracts, and end users would only interact with the ALCs. Utility contracts would provide utility functions, and CMCs would be responsible for keeping track of all the contracts in the system. Implementing such a comprehensive system was beyond the scope of this project, but it would provide more flexibility in upgrading and extending the system, and might be worth exploring in the future.

The Future of Ethereum

During the previous peak in usage of the Ethereum network during the 2017-2018 adoption rally, the network became congested due to the demand, causing a significant raise in gas cost.[44] At the same time, the price of ETH climbed to more than €1,100.[45] An ETH price five times what it is today, and a gas cost increase of approximately four times would cause a twenty-fold increase in actual transaction costs, rendering this service's costs no longer superior to the industry examples described previously.

[43] "Tutorials | Solidity 1: The Five Types Model" 7 Jun. 2018, https://github.com/monax/legacy-docs/blob/master/solidity/solidity_1_the_five_types_model.md Accessed 6 Jun. 2020.

[44] "Ethereum Gas Price Analysis - Onther - Medium." 17 Oct. 2018, https://medium.com/onther-tech/ethereum-gas-price-analysis-b70080e2e0d7. Accessed 6 Jun. 2020.

[45] "ETH EUR Kraken Historical Data - Investing.com." https://www.investing.com/crypto/ethereum/eth-eur-historical-data. Accessed 6 Jun. 2020.

The development of Ethereum version 2.0 should address these issues, hopefully by the next time a surge in demand occurs. With the new version, Ethereum will replace the *proof-of-work* (POW) algorithm with the much more energy efficient *proof-of-stake* algorithm, and this change, among other improvements, should reduce gas cost of transactions and contract interactions significantly.[46]

[46] "A Comprehensive view of Ethereum 2.0 (Serenity) - Medium." 14 Nov. 2019, https://medium.com/swlh/a-comprehensive-view-of-ethereum-2-0-serenity-5865ad8b7c62. Accessed 6 Jun. 2020.

References

1 - Nakamoto, Satoshi. (2008, October). *Bitcoin: A Peer-to-Peer Electronic Cash System*. Originally published in Cryptography mailing list at metzdowd.com. Retrieved from https://bitcoin.org/bitcoin.pdf.

2 - Szabo, Nick. (1997). *Formalizing and Securing Relationships on Public Networks*. Originally published in digital journal First Monday. Retrieved from https://nakamotoinstitute.org/formalizing-securing-relationships/.

3 - Buterin, Vitalik. (2013, December). *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform*. Originally published on Buterin's blog www.vbuterin.com. Retrieved from https://ethereum.org/whitepaper/.

4 - Hitchens, Rob. (2017, February). *Solidity CRUD*. Medium.com. Retrieved from https://medium.com/robhitchens/solidity-crud-part-1-824ffa69509a and https://medium.com/robhitchens/solidity-crud-part-2-ed8d8b4f74ec.

5 - Atzei, Nicola & Bartoletti, Massimo & Cimoli, Tiziana. (2017, March) *A Survey of Attacks on Ethereum Smart Contracts*. Published in *Principles of Security and Trust* by Springer Berlin Heidelberg.

6 - Madeira, Antonio. (2019, March). *The Dao, the Hack, the Soft Fork and the Hard Fork*. CryptoCompare.com. Retrieved from https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/.

7 - Nuvreni, Juan. (2019, November). *Ethereum 2.0 (Serenity): Bringing the World's Super Computer to Life.* BitPrime.co.nz. Retrieved from https://www.bitprime.co.nz/blog/ethereum-2-0-serenity-bringing-worlds-super-computer-life/