ERC777

A New Advanced Token Standard For The Ethereum Blockchain

Master's Thesis submitted to the

Faculty of Informatics of the *Università della Svizzera Italiana, Switzerland*

in partial fulfillment of the requirements for the degree of

Master of Science in Informatics

presented by

Jacques Dafflon

under the supervision of

Prof. Cesare Pautasso

co-supervised by

Thomas Shababi

June 2018

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Jacques Dafflon
Neuchâtel, 20 June 2018

Abstract

Ethereum decentralized computing Ethereum is decentralized platform running on a custom built blockchain launched on July 30th 2015. Unlike more traditional crypto-currencies, Ethereum is a computing platform with a Turing-complete programming language used to create and execute arbitrary pieces of code known as smart contracts.

In this thesis describes how the power of smarts contracts is leveraged to have fast transactions despite the slowness of the blockchain.

Acknowledgements

ACKS

Contents

Contents **vii**

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Description Of The Work . . . 1

2 Ethereum, A Decentralized Computing Platform **3**

2.1 The Ether Currency And Gas . . . 3

2.1.1 Computing Fees . . . 3

2.2 Ethereum Accounts . . . 4

2.3 Transactions And Messages . . . 4

2.4 The Ethereum Virtual Machine . . . 5

2.5 Solidity . . . 5

2.5.1 View Functions . . . 5

3 Tokens And Standardization **7**

3.1 Definition of a token . . . 7

3.2 Standardization . . . 8

3.3 Ethereum Improvement Proposals And Ethereum Request For Comments . . . 8

4 ERC20 Token Standard **9**

4.1 The First Token Standard . . . 9

4.2 Transferring ERC20 Tokens . . . 10

4.3 Strength And Weaknesses Of ERC20 . . . 14

4.3.1 Locked Tokens . . . 14

4.3.2 Approval Race Condition . . . 14

4.3.3 Absence Of burning . . . 15

4.3.4 Optional `decimals` function. . . . 16

4.3.5 A Retroactive Standard . . . 16

5 ERC777, A New Advanced Token Standard For Ethereum Tokens 17
5.1 Operators 17
5.1.1 Default Operators 18
5.1.2 Authorizing And Revoking Operators 18
5.2 Hooks 19
5.2.1 Preventing Locked Tokens 19
5.2.2 Location Of The Hooks 19
5.3 Sending Tokens 20
5.4 Minting Tokens 20
5.5 Burning Token 20
5.6 View Functions 20
5.7 Decimals 21
5.8 Compatibility 21
5.8.1 Notifying The Recipient 21
5.8.2 ERC20 Backward Compatibility 21

6 ERC820, Pseudo-introspection Registry contract 23

7 Competing Token Standards 25
7.1 ERC227 25
7.2 ERC827 25
7.3 ERC995 25

8 The State Of Tooling 27
8.1 Compilation 27
8.2 Testing and Coverage 27
8.3 Documentation 27
8.4 Missing Tools 27

9 Future Research and Work 29

10 Conclusion 31

Glossary 33

Figures

4.1 Standad ERC20 `transfer` between two regular accounts: Alice and Bob. 10
4.2 Standad ERC20 `transferFrom` between a regular account Alice and a contract account Carlos. 12
4.3 Sequence of calls for the `approve`/`transferFrom` attack. 15

Tables

Chapter 1

Introduction

1.1 Motivation

Ethereum is a new blockchain inspired by Bitcoin, with the design goal of abstracting away transaction complexity and allowing for easy programatic interaction through the use of a Virtual Machine. One issue is the design of ERC20, the token standard. The way to transfer tokens to an externally owned address or to a contract address differ and transferring tokens to a contract assuming it is a regular address can result in losing those tokens forever.

The new ERC777 token standard solves those problems and offer new powerful features which facilitates new interesting use cases for tokens.

1.2 Description Of The Work

This thesis starts by explaining the Ethereum ecosystem and the concept of tokens on the blockchain as well as the application of tokens in Ethereum, the ERC20 token standard and its limitations. It is followed by a detailed description of the new ERC777 standard [Dafflon et al., 2017] for tokens and the ERC820 Pseudo-introspection registry[Baylina and Dafflon, 2018], developed as part of this thesis. This includes how the issues of ERC20 are solved, the improvements brought by ERC77, the efforts made to advertise the standard to the community and how community's reception and feedback was taken into account to developed the standard. Finally we provide an analysis of the ERC223 and ERC827 token standards proposals—which are alternatives to ERC777—and their drawbacks.

Chapter 2

Ethereum, A Decentralized Computing Platform

The Ethereum network is a decentralized computing platform. As described in its the white paper, Ethereum "[...] is essentially the ultimate abstract foundational layer: a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions" [Buterin, 2013]. This differentiate it from Bitcoin which is a trustless peer-to-peer version of electronic cash and lacks a Turing-complete language.

2.1 The Ether Currency And Gas

The Ethereum still includes its own built-in currency named ether akin to Bitcoin. It "[...] serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees" [Buterin, 2013]. The currency comes with different denominations defined. The smallest denomination is a wei—named after the computer scientist and inventor of b-money, Wei Dai. An ether is defined as 10^{18} wei. In other words a wei represents 0.000000000000000001 ethers. The wei denomination is used for technical discussions. Most tools, libraries and smart contracts use wei and the values are only converted to ether or some other denomination for the end-user.

2.1.1 Computing Fees

The fees is part of the incentive mechanism as in Bitcoin. The main difference is the way the fees are expressed and computed. In Bitcoin the fees are fixed and set as the difference between the input value and the output value. Because transactions on the Ethereum network execute code of a Turing-complete language, the fee is defined differently "[...] to prevent accidental or hostile infinite loops or other computational wastage in code" [Buterin, 2013]. A transaction define two fields `STARTGAS` and `GASPRICE`. The `STARTGAS`—also referred as just `gas` or `gasLimit`—is the maximum amount of gas the transaction may use. The `GASPRICE` is the fee the sender will pay per unit of gas consumed. Essentially, the fees is a limitation on the Turing-completeness. While the language is Turing-complete the execution of the program is limited in its number of steps. In essence, fees are not only a part of the incentive mechanism but are also an anti-spam measure as every extra transaction is a burden on everyone in the network, and it would be effectively free to grief the network if there were no fees.

A computational step cost roughly one unit of gas. This is not exact as some steps "cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state" [Buterin, 2013]. A cost of five units of gas per byte is also applied to all transactions.

Another advantage of not tightly coupling the cost of execution with a currency—e.g. set the cost of a computation step to three wei—is to dissociate the execution cost of a transaction and the fluctuation in value of ether with respect to fiat currencies. If the price of ether increases exponentially with respect to a currency such as the dollar, a fixed price per computational step may become prohibitive. The `GASPRICE` circumvent this issue. While the amount of gas consumed by the transaction will remain constant, the price for the gas can be reduced.

2.2 Ethereum Accounts

There are two types of accounts on the Ethereum network, externally owned accounts—commonly referred to as regular accounts—and contract accounts. A regular account is an account controlled by a human who holds the private key needed to sign transactions. In contract, a contract account is an account where no individual knows the private key. The account can only send its ether and call function of other accounts through its associated code. While an account is defined as "having a 20-byte address and state transitions being direct transfers of value and information between accounts" [Buterin, 2013], the words "account" and "address" are often used interchangeably. Nonetheless, to be exact, an account is defined in the white paper [Buterin, 2013] as a set of four fields:

Nonce A counter used to make sure each transaction can only be processed once

Balance The account's current ether balance

Code The account's contract code, if present (for contracts)

Storage The account's permanent storage (empty by default)

2.3 Transactions And Messages

Ethereum makes a distinction between a transaction and a message. A transaction is a signed data packet only emitted from a regular account. This packet contains the address of a recipient, a signature to identify the sender, the amount of ether sent from the sender to the recipient a data field—which is optional and thus may be empty and the gas price and gas limits whose meaning is explained in section 2.1.1.

A message is defined as a "virtual objects that are never serialized and exist only in the Ethereum execution environment" [Buterin, 2013]. A message contains the sender and recipient, the amount of ether transfer with the message from the sender to the recipient, an optional potentially empty data field and a gas limit.

Transactions and messages are very similar. The difference is that a transaction comes from a regular account only and a message comes from contract. A transaction can call a function of a contract which in turn can create a message and call another function, either on itself or on another contract, using the `CALL` and `DELEGATECALL` opcodes. The gas used for messages comes from the transaction which triggered the call.

2.4 The Ethereum Virtual Machine

Ethereum is a decentralized computing platform. In other words alongside a blockchain, Ethereum provides a Turing-complete language and the Ethereum Virtual Machine (EVM), a virtual machine able to interpret and execute code. This code "is written in a low-level, stack-based bytecode language, referred to as"Ethereum virtual machine code" or "EVM code"[Buterin, 2013]. This bytecode is represented by a series of bytes. Code execution simply consist of setting an instruction pointer at the beginning of the bytecode sequence, perform the operation at the current location of the point, and then increment the instruction pointer to the next byte. This is repeated forever until either the end of the bytecode sequence is reached, an error is raised or a `STOP` or `RETURN` instruction is executed.

The operations can perform computations and interact with data. There are three kinds of mediums to store data. First, there is a stack. This a commonly know abstract data type in computer science. Data can be added by using a push operation which adds the data on top of the stack. Mutually, the data can then be removed with a pop operation which removes and return the data from the top of the stack. Essentially, the stack is known as a Last-In-First-Out (LIFO) data structure meaning the last value pushed (added) is the first value popped (taken). Secondly there is a memory, which is an ever-expandable array of bytes. Those kinds of storage are both non-persistent storage. Within the context of Ethereum, this translates to this data only being available within the call or transaction and not being permanently stored on the blockchain. The third and last kind of storage is commonly referred to as "storage" is a permanent key/value store intended for long-term storage.

In addition to those types of storage, the code may access the block header data, and the incoming transaction's sender address, value, and data field.

2.5 Solidity

2.5.1 View Functions

View functions in Solidity are defined as function which do not modify the state. As defined in the Solidity documentation [Solidity documentation], modifying the state implies one of:

1. Writing to state variables.
2. Emitting events.
3. Creating other contracts.
4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked view or pure.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

Chapter 3

Tokens And Standardization

3.1 Definition of a token

A token in a generic term is a digital asset. A token can essentially represent any asset which is fungible and tradable. Tokens are built on top of an existing blockchain and with its Turing-complete language and popularity, Ethereum is a prime candidate as a platform to build tokens on top of. The concept of tokens is in fact described in the Ethereum white paper as "[having] many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization" [Buterin, 2013]. Examples of tokens include EOS, TRON, Status, Aragon or Gnosis.

Tokens are the result of certain types of smart contracts which maintain a ledger on top the Ethereum blockchain and with the goal of acting like a "coin". Internally this smart contract simply holds a mapping from addresses to balances. The balances are expressed with unsigned integers. This design choice is similar to ethers which themselves internally are expressed as wei. It also comes from the fact that the Solidity language does not fully support floating point numbers. The smart contract then exposes functions to let user acquire tokens—known as minting—destroy tokens—known as burning—and most importantly to let token holders transfer their tokens. From a business perspective, a token is the possibility for a company to issue shares, securities or any form of accounting unit; even their own currency which the company has control over. Many companies offer services which can be purchased only using their tokens. Based on this economic principle, comes the neologism: Initial Coin Offering or ICO. An ICO is a process where a company will sell a limited quantity of their tokens for a fixed price before their product is finalized. This is for a start-up a mean to raise funds on their own without having to go through the vetting process traditionally required by venture capitals and banks. An ICO is usually done through a smart contract which will trade tokens for ethers at specific times and for a certain price. This allows the start up to raise some capital and the investors to potentially gain a profit by buying tokens at a discount. There is of course the risk that the start-up fails and the tokens become worthless.

3.2 Standardization

With many start-ups creating tokens to make initial coin offerings, building Decentralized Applications (DApps) and providing various services both on-chain and off-chain to use theses tokens; the need for a standardized way to interact with said tokens arose rapidly. A standard for tokens allows wallet—holding a user's private key—to easily let the use interact with both their ether and a wide collection of their token easily. It allows any smart contract—wether it is a wallet or a DApp—to effortlessly received, hold and send tokens. Smart contract are immutable which makes them notoriously hard to update. Typically, any update is done by replacing an existing smart contract with a new one at a different address with a copy of the data. Any off-chain infrastructure must then point to the address of this new contract. Updating a smart contract to handle a different way of interacting with a new and specific token would be an impossible task. Having a standard which defines an interface to interact with tokens allows DApps and wallet to instantly be compatible with any existing and future token which complies with the the standard.

3.3 Ethereum Improvement Proposals And Ethereum Request For Comments

Blockchain projects in general, including Ethereum, are ecosystems which tend to be available as open-source software. Their projects are community oriented where anyone is invited to participate and contribute. To distribute source code and organize contributions the Ethereum Foundation relies on their Github organization account. One of the repository they maintain is the Ethereum Improvement Proposals or EIPs. This repository, available at github.com/ethereum/EIPs "[...] describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards" [EIPs, homepage]. This includes the Ethereum Requests for Comments track which includes "Application-level standards and conventions, including contract standards such as token standards (ERC20), name registries (ERC137), URI schemes (ERC681), library/package formats (EIP190), and wallet formats (EIP85)" [EIPs]. This is the track where the current standard for tokens is defined and where any proposal for new token standards take place. New standards are submitted by opening a pull request—previously an issue—containing a description of the standard proposal and following the provided template. This template states: "Note that an EIP number will be assigned by an editor" [EIP-X]. In practice and historically the number associated to an EIP is the number of the initial issue or pull request which started the standard. This as a matter of fact applies to all the EIPS discussed in this paper.

Chapter 4

ERC20 Token Standard

The first—and so far only—standard for tokens on the Ethereum blockchain is ERC20. There have been many new token standards and extensions or improvements to ERC20 suggested over time. Not including ERC777—the standard detailed in this paper—such proposals include the following standards: the ERC223 token standard, the ERC827 Token Standard (ERC20 Extension), the ERC995 Token Standard, the ERC1003 Token Standard (ERC20 Extension), and changes to ERC20 such as: Token Standard Extension for Increasing & Decreasing Supply (ERC621), Provable Burn: ERC20 Extension (ERC661), Unlimited ERC20 token allowance (ERC717), Proposed ERC20 change: make 18 decimal places compulsory (ERC724), Reduce mandatority of Transfer events for void transfers (ERC732), Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack (ERC738), Extending ERC20 with token locking capability (ERC1132).

4.1 The First Token Standard

The ERC20 standard was created on November 19th 2015 as listed on the EIPs website under the ERC track [see Foundation, 2018a, ERC track]. A standard for tokens must define a specific interface and expected behaviors a token must have when interacted with. This allows wallets, DApps and services to easily interact with any token. It defines a simple interface which lets anyone transfer their tokens to other address, check a balance, the total supply of tokens and such. Specifically it defines nine functions a token must implement: `name`, `symbol`, `decimals`, `totalSupply`, `balanceOf`, `transfer`, `transferFrom`, `approve`, `allowance` as well as two events which must be fired in particular cases: `Transfer` and `Approval`.

The `name`, `symbol` are optional functions which fairly basic and easy to understand. They return the name and the symbol or abbreviation of the token. Considering the Aragon token as an example, the `name` function returns the string `Aragon Network Token` and the `symbol` functions returns `ANT`. Another somewhat harder to understand optional function is `decimals`. This function returns the number of decimals used by the token and thus define what transformation should be applied to any amount of tokens before being displayed to the user or communicated to the token contract. As previously explained, the balances and amounts of tokens handled by the token contracts are (256 bits) unsigned integers. Therefore the smallest fractional monetary unit is one. For some—or many—tokens, it makes more sense to allow smaller fractions. The `decimals` function return the number of decimals to apply to any amount passed to or returned by the token contract. Most tokens simply follow Ether—which uses eighteen decimals—and use eighteen decimals as well. Another decimals value used is zero. A token with zero decimals can make sense when a token represent an entity which is not divisible—such as a physical entity. Altogether those functions are optional and purely cosmetic. The most important function being `decimals` as any misuse will show an incorrect representation of tokens and thus of value.

The `totalSupply` and `balanceOf` are also `view` functions. Simply put, they do not modify the state of the token contract, but only return data from it. This behavior is similar to what one can expect from getter functions in object oriented programming.

The `totalSupply` function returns the total number of tokens held by all the token holders. This number can either be constant or variable. A constant total supply implies the token contract is created with a limited supply of tokens with neither the ability to mint tokens nor the ability to burn. A variable total supply, on the other hand, signifies that a the token contract is capable of minting new tokens or burning them or both.

The `balanceOf` function takes an address as parameter and returns the number of tokens held by that address.

4.2 Transferring ERC20 Tokens

The `transfer` and `transferFrom` functions are used to move tokens across addresses. The `transfer` function takes two parameters, first the address of the recipient and secondly the number of tokens to transfer. When executed, the balance of the address which called the function is debited and the balance of the address specified as the first parameter is credited the number of token specified as the second parameter. Of course, before updating any balance some checks are performed to ensure the debtor has enough funds.

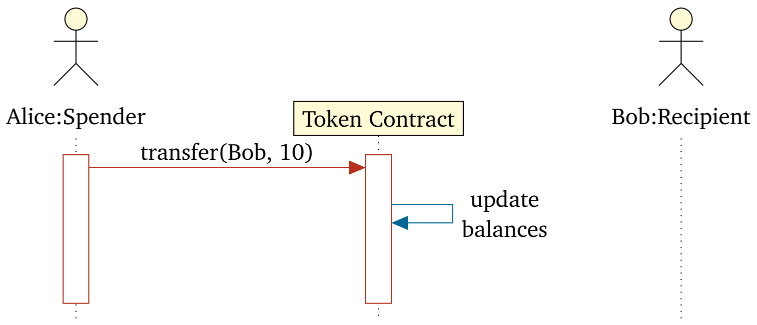

Figure 4.1. Standad ERC20 `transfer` between two regular accounts: Alice and Bob.

As seen on figure 4.1 when performing a transfer, the spender emits a transaction which calls the token contract and update the balances accordingly. The recipient is never involved in the transaction. The logic to update the balance is entirely done within the token contract, in the `transfer` function.

Examples of the implementation details to update the balances are shown in listings 4.1 and 4.2.

```
/**
```

```
2  * @dev Transfer token for a specified address
3  * @param _to The address to transfer to.
4  * @param _value The amount to be transferred.
5  */
6  function transfer(address _to, uint256 _value) public returns (bool) {
7    require(_value <= balances[msg.sender]);
8    require(_to != address(0));
9
10   balances[msg.sender] = balances[msg.sender].sub(_value);
11   balances[_to] = balances[_to].add(_value);
12   emit Transfer(msg.sender, _to, _value);
13   return true;
14 }
```

Listing 4.1. OpenZepplin's implementation of ERC20's transfer function.

The implementation of the `transfer` function in the listing 4.1 shows—on lines 7 and 8—the conditions checked before effectually performing the transfer and update of the balances—on lines 10 and 11. The solidity instruction `require` evaluates the condition it is passed as parameter. If it is true it will continue with the execution, otherwise it will call the REVERT EVM opcode which stops the execution of the transaction without consuming all of the gas and revert the state changes.

The first check ensure that the token holder—here referred as the sender—does not try to send a number of tokens higher than its balance. The variable `msg.sender` is a special value in Solidity which holds the address of the sender of the message for the current call. In other words, `msg.sender` is the address which called the `transfer` function.

The second checks ensure that the recipient—defined in the parameter `_to`—is not the zero address. The notation `address(0)` is a cast of the number literal zero to a 20 bits address. The zero address is a special address. Sending tokens to the zero address is assimilated to burning the tokens. Ideally the balance of the zero address should not be update in this case. This is not always the case, tokens such as Tronix are held by the zero address. A quick look at their implementation shown in listing 4.2 of the transfer function shows there is no check to ensure the recipient is not the zero address. Note that the `validAddress` modifier only verifies the `msg.sender` or in other words, the spender, not the recipipent.

```
modifier isRunning {
    assert (!stopped);
    _;
}

modifier validAddress {
    assert(0x0 != msg.sender);
    _;
}

// ...

function transfer(address _to, uint256 _value)
    isRunning validAddress returns (bool success)
```

```
{
    require(balanceOf[msg.sender] >= _value);
    require(balanceOf[_to] + _value >= balanceOf[_to]);
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
    Transfer(msg.sender, _to, _value);
    return true;
}
```

Listing 4.2. Tronix transfer function.

The `transferFrom` function is the second function available to transfer tokens between addresses. It's use is depicted in figure 4.2. It takes three parameters the debtor address, the creditor address and the number of tokens to transfer.

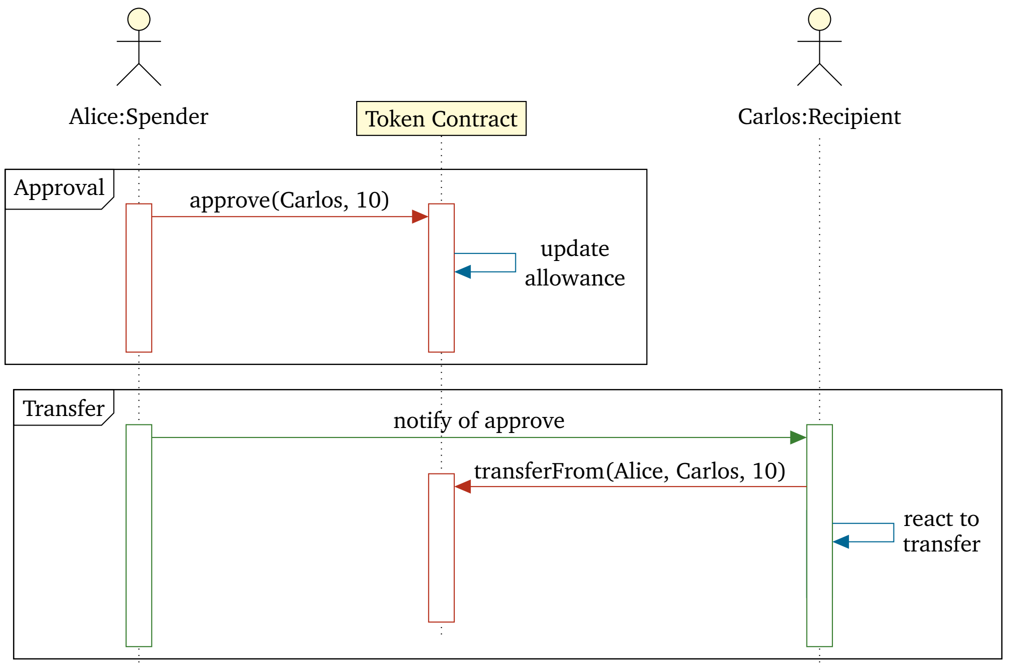

Figure 4.2. Standad ERC20 `transferFrom` between a regular account Alice and a contract account Carlos.

The reason for the existence of this second function to transfer tokens is contracts. Contracts usually need to react when receiving tokens. When a normal `transfer` is called to send tokens to a contract, the receiving contract is never called and cannot react. Contracts are also not able to listen to events, making it impossible for a contract to react to a `Transfer` event. The `transferFrom` lets the token contract transfer the tokens from someone else to itself or others. At first glance this appears to be insecure as it lets anyone withdraw tokens from any address. This is where the `approve` and `allowance` functions come into play. The specification for the `transferFrom` function state that "[t]he function SHOULD `throw` unless the `_from` account has deliberately authorized the sender of the message via some mechanism" [Vogelsteller and Buterin, 2015]. The `approve` function the standard mechanism to authorize a sender to call transferFrom. Consider an ERC20 token, a regular user Alice and a contract Carlos. Alice wishes to send five tokens to Carlos to purchase a service offered by Carlos. If she uses the transfer function, the contract will never be made aware of the five tokens it received and will not activate the service for Alice. Instead, Alice can call approve to allow Carlos to transfer five of Alice's tokens. Anyone can then call allowance to check that Alice did in fact allow Carlos to transfer the five tokens from Alice's balance. Alice can then call a public function of Carlos or notify off-chain the maintainers of the Carlos contract such that they can call the function. This function of Carlos can call the transferFrom function of the token contract to receive the five tokens from Alice.

The internals of the transferFrom function are similar to those of the transfer function. The main differences are that the debtor address is not msg.sender but the value of the _from parameter, and there is—in most cases—an additional check to make sure whoever calls transferFrom is allowed to withdraw tokens of the _from address. of course, the allowed amount is updated as well for a successful transfer. The listing 4.3 shows OpenZepplin's implementation of the function, which performs the allowance check on line 16 and the update of the allowance on line 21. The balance updates is similar to the transfer function from listing 4.1, except that the parameter _from is used instead of msg.sender as the debtor.

```
 1  /**
 2   * @dev Transfer tokens from one address to another
 3   * @param _from address The address which you want to send tokens from
 4   * @param _to address The address which you want to transfer to
 5   * @param _value uint256 the amount of tokens to be transferred
 6   */
 7  function transferFrom(
 8    address _from,
 9    address _to,
10    uint256 _value
11  )
12    public
13    returns (bool)
14  {
15    require(_value <= balances[_from]);
16    require(_value <= allowed[_from][msg.sender]);
17    require(_to != address(0));
18
19    balances[_from] = balances[_from].sub(_value);
20    balances[_to] = balances[_to].add(_value);
21    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
22    emit Transfer(_from, _to, _value);
23    return true;
24  }
```

Listing 4.3. OpenZepplin's implementation of ERC20's transferFrom function.

4.3 Strength And Weaknesses Of ERC20

Overall the ERC20 token standard was kept simple in its design. Hence the standard results in simple token contracts. This is one of the upside of the standard. Token contracts can be kept short and simple which makes them easy and cheap to audit. This is especially important as an insecure contract will lose its value extremely quickly and good smart contract auditors are expensive and have little availability.

At the other end of the spectrum however, this translates to a higher burden on the user, applications and wallets interacting with the tokens.

4.3.1 Locked Tokens

One of the largest issue is the distinction all token holders must make when transferring tokens regarding the type of recipient. There are no issues if the recipient is a regular account, `transfer` just works and calling `approve` with the correct amount and let the recipient call `transferFrom`. The user experience in this case is somewhat suboptimal as it requires off-chain communication, two transactions, and the recipient has two pay the gas for the second transaction. Nonetheless the intended goal is achieved and the transfer from the spender to the recipient is achieved.

The same cannot be said if the recipient is a contract account. When using `transfer` to send tokens to a contract. The spender initiates the transfer and only communicates with the token contract the recipient is never notified—as previously shown in figure 4.1. The result is that while the token balance of the receiving contract is increased, that contract may never be able to use and spend the tokens it received—this situation is commonly referred to as "locked tokens". A simple proof is the Tronix contract whose `transfer` function was discussed before. A rapid look at the token balance of the Tronix contract—deployed at itself shows a balance of 5'504'504.3514 TRX as of August 8th 2018. With an exchange rate of \$0.0272, this represent a value of just a little under 150,000 US dollars. By analyzing the code, one can see there are no functions which would allow the contract to spend those tokens. There are of course many more similar examples of such scenarios where people sent tokens either to the token contract or to some other contract by mistake and the amounts add up quickly

4.3.2 Approval Race Condition

By abusing the Application Binary Interface (ABI) of ERC20, an attacker can trick its victim into approving more tokens for the attacker to spend than intended. This attack was revealed on November 29th 2018. Essentially, it takes advantage of two of ERC20's functions: `approve` and `transferFrom`. Because this is an issue with the logic in the standard, all ERC20-compliant implementations are affected. This attack works as follow, as described in the original paper [Vladimirov and Khovratovich, 2016]:

1. Alice allows Bob to transfer N of Alice's tokens ($N > 0$) by calling the `approve` function on the token smart contract, passing Bob's address and N as function arguments.
2. After some time, Alice decides to change from N to M ($M > 0$) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve` function again, this time passing Bob's address and M as function arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls `transferFrom` function to transfer N Alice's tokens somewhere.

4. If Bob's transaction is executed before Alice's transaction, then Bob will successfully transfer N of Alice's tokens and will gain an ability to transfer another M tokens.
5. Before Alice noticed that something went wrong, Bob calls the `transferFrom` function again, this time to transfer M of Alice's tokens.

So, Alice's attempt to change Bob's allowance from N to M ($N > 0$ and $M > 0$) made it possible for Bob to transfer $N + M$ of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

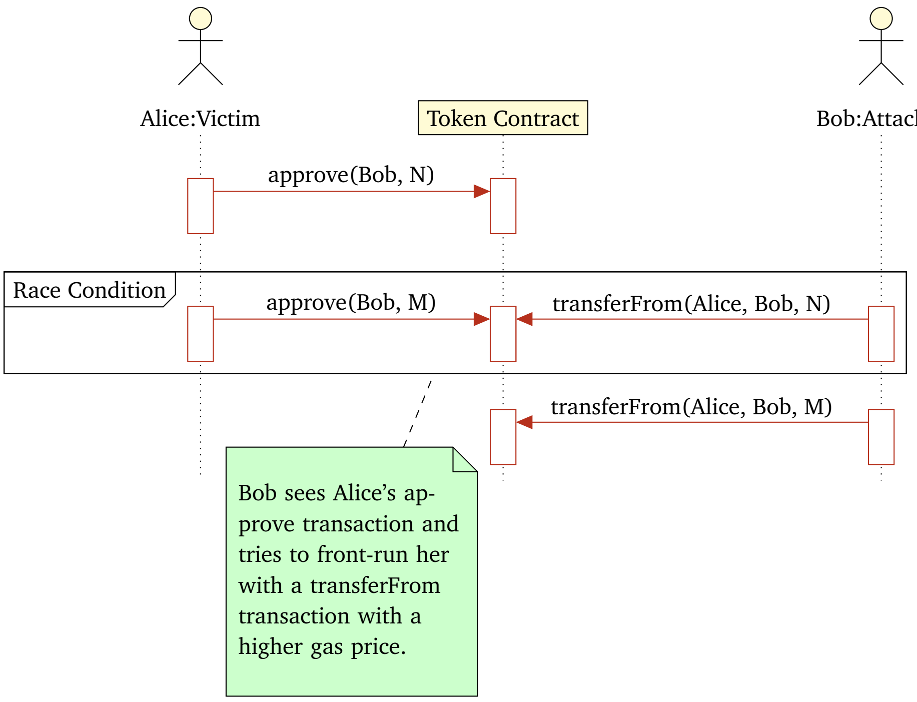
Alice:Victim
Token Contract
Bob:Attacker
approve(Bob, N)
Race Condition
approve(Bob, M)
transferFrom(Alice, Bob, N)
transferFrom(Alice, Bob, M)
Bob sees Alice's approve transaction and tries to front-run her with a transferFrom transaction with a higher gas price.

Figure 4.3. Sequence of calls for the `approve`/`transferFrom` attack.

4.3.3 Absence Of burning

The ERC20 standard defines the behavior for minting new tokens. Namely, "[a] token contract which creates new tokens SHOULD trigger a Transfer event with the `_from` address set to `0x0` when tokens are created" [Vogelsteller and Buterin, 2015]. Unfortunately the standard does not go further, nothing is specified regarding the balance of `0x0` or the `totalSupply` for example.

Furthermore, the standard does not contain any specification about burning tokens. Sending to the `0x0` address is commonly assumed to represent burning—not to be confused with voluntary locking where the tokens are sent to some other address made of a repeating and non-random looking pattern such as `0x1111111111111111111111111111111111111111`. While "sending to `0x`" is a perfectly reasonable abstraction to represent a burn of tokens, it needs to be clearly defined. Should the balance of `0x0` be incremented? If yes, should the total supply remain the same or should it ignore the balance of `0x0` and be decreased? Should a `Transfer` event with the `to` address set to `0x0` be emitted? Can the tokens be burned using a `transfer` or `transferFrom` call with `to` set to `0x0` or should specific function be used to burn the tokens?

Out of all the questions above, most tokens tend to emit `Transfer` events with the `to` address set to `0x0`. The remaining questions are solved differently for various tokens. Multiple mutually exclusive solution may be acceptable. However in some cases, some solutions may be preferable over others. As an example, most of the smart contracts are written in Solidity where an uninitialized variable of type `address` has a value of zero (`0x0`). On the off-chance that the value passed as the `to` parameter to a `transfer` call is uninitialized, then if the token contract allows burning via `transfer`, this will result in an unintentional burn of the tokens. In such a scenario, it may be preferable to revert the transaction instead and expose a specific function to burn tokens instead.

4.3.4 Optional `decimals` function.

As specified in ERC20, the `decimals` function is:

> OPTIONAL - This method can be used to improve usability, but interfaces and other contracts MUST NOT expect these values to be present [Vogelsteller and Buterin, 2015].

In practice this result in ERC20 compliant tokens which do not implement the `decimals` function. While this may be reasonable, there is no default value defined in the standard in this scenario. This is a serious issue because if the token contract holds a balance of 2,000,000,000,000,000,000 tokens, the actual balance displayed to the user may range anywhere from 2,000,000,000,000,000,000 all the way down to 2. Common values returned by `decimals` are 18 (equivalent to ether), 8 which is the value used in Bitcoin or 0 for indivisible tokens. Obviously this is problematic, especially when the token holds a value. There is no constraint in the ERC20 standard to enforce a constant `decimals` value. Thankfully, there is—to our knowledge—no token having a variable `decimals`.

Vitalik Buterin tried to solve this impression [**?**] but the issue is still ongoing today.

4.3.5 A Retroactive Standard

While the drawbacks of ERC20 mentioned above appear to be poor design, an important factor when writing the standard was that tokens were already being used before the standard was finalized. The standard was therefore written in such a way that those first “ERC20” token would remain ERC20-compliant.

> This resulted in things being SHOULD instead of MUST because not all ERC20 tokens followed the rule, so setting it as MUST would have [resulted] in some tokens that were already known commonly as ERC20 tokens suddenly not being ERC20 [Zoltu, 2018].

The result is that it makes it hard to modify and improve the standard without breaking backwards compatibility with existing tokens. This is also one of the main reason behind the need for a new and better token standard.

Chapter 5

ERC777, A New Advanced Token Standard For Ethereum Tokens

ERC777 is a new advanced token standard for Ethereum tokens. It is the result of the work described in this paper and made in collaboration with Jordi Baylina from the White Hat Group and Giveth.

The standard describes three central mechanisms: sending tokens, minting tokens and burning tokens. Those mechanisms are performed by a specific role—an operator—which is also defined in the standard. These mechanisms take advantage of hooks—specific functions which are called to notify and/or control the debit or credit of tokens. Lastly, ERC777 includes extra constraints for backwards compatibility with ERC20.

Creating a new standard requires careful consideration. Many aspects had to be considered such as security, usability, compatibility with the existing ecosystem and backward compatibility with existing ERC20 infrastructures. All things considered, ERC777 brings many enhancements including data associated with transactions, operators, hooks and backward-compatibility with ERC20 which address the previously mentioned considerations.

5.1 Operators

An operator is a specific role which must be defined first, in order to properly understand the three mechanisms described below. In one sentence:

> An operator is an address which is allowed to send and burn tokens on behalf of another address [Dafflon et al., 2017].

On top, of this core definition, constraints are defined and applied for all operators. First, every address is always an operator for itself. This right is not revocable. Second, any address–regular accounts or contract—is allowed to authorize and later revoke other addresses as their operators. Therefore some accounts may have their token funds managed by another party. Ideally, operators are intended to be contracts whose code may be audited. As a result, users can authorize a contract as their operator without the fear of the operator withdrawing all their tokens. Evidently this implies users have previously verified the code of the operator and they have convinced themselves the code operator does not include vulnerabilities and is not able to withdraw all the funds. Examples of such operator contracts include payment or cheque processors, Decentralized Exchanges (DEXs), subscription managers and automatic payment systems.

There are also interesting scenarios which leverage hooks to authorize regular accounts as operators whilst only letting them spend tokens according to specific rules.

5.1.1 Default Operators

All addresses are automatically and irrevocably operators for themselves—and may explicitly authorize any other address(es) as operator(s). Additionally, any token contract may define a set of operators at creation/deployment time which are implicitly authorized for all token holders. This feature allows token designers to offer additional features specific to their token—with a modular design—to let their users move their funds more seamlessly/in a more integrated fashion. It's worth noting that a token contract which enables default operators would implicitly require that these operators are included in any review of the token contract. Taking inspiration from the previous examples for operators. If a token is used a form of payment for subscription. The company behind the service may be interested to not only create the token but an operator to directly and regularly levy the subscription fee. Since the use—and therefore the value—of the token are based on this subscription service, it is logical to authorize the subscription operator by default. Obviously, default operators can be revoked by the token holder and a token contract must not be able to change the list of default operators after the contract is created.

5.1.2 Authorizing And Revoking Operators

Authorizing operators is the process where an address authorizes another address as its operator. An address may authorize many operators at the same time. However only the token holder can authorize operators for itself.

This last constraint is extremely important for contract as it implies that one must correctly implement some logic to let the contract authorize the operators it needs. Essentially only the contract itself is allowed to set its operators. For a contract which does not expose a function to perform arbitrary calls, it must implement one or more call to authorize operators in the constructor or some other function. Otherwise, the contract will never be able to authorize any operator.

ERC777 defines a specific function to authorize an operator: `authorizeOperator`. This function takes the address of the operator as parameter and authorizes it for the address which initialized the call (`msg.sender`). The standard requires the implementation of this function but it does not however constrain the authorization mechanism to this function. In other words, the token may define other mechanisms to authorize operators as long as those mechanisms are compliant. For example they must emit the `AuthorizedOperator` event with the correct data.

Similarly, ERC777 defines a specific function to revoke an operator: `revokeOperator`. This function takes the address of the operator as parameter and revokes it as an operator for the address which initialized the call (`msg.sender`). The standard requires the implementation of this function but it does not however constrain the revocation mechanism to this function. Essentially, the token may define other mechanisms to revoke operators as long as those mechanisms are compliant. For example they must emit the `RevokedOperator` event with the correct data.

5.2 Hooks

A hook is a specific function which is called to notify and let the accounts control the debit or credit of tokens. Specifically, ERC777 defines two hook functions `tokensToSend` to notify of the debit of tokens and `tokensReceived` to notify of the credit of tokens. Furthermore, if the recipient is a contract, the `tokensReceived` hook must be called—to notify of the credit and give an opportunity for the receiving contract to react and to prevent locking the tokens.

In any case, both the account from which the tokens are debited and the account to which the tokens are credited can implement hooks which revert if a condition is met. Thus providing greater control to accounts over which token they wish to send or receive.

5.2.1 Preventing Locked Tokens

Locked tokens, as explained in section 4.3.1, refers to tokens held by a contract which is not aware of the fact and is unable to spend said tokens as these contracts do not have the functionality to call the token contract and have no upgrade or migration system in place—often as a design choice. The `tokensReceived` hook gives the chance for contracts to be notified of any credit and react to it. This hook is extremely important from a safety standpoint when the recipient is a contract, therefore ERC777 stipulates that it is mandatory for contracts to register a `tokensReceived` hook in order to receive tokens. This is the only scenario where a hook is required. Sending tokens to a regular account will never result in locked token, providing the person behind the account has the private key. Some people explicitly lock tokens by sending them to addresses composed of specific patterns for which it is unlikely anyone has the private key. Example of such accounts include `0x1111111111111111111111111111111111111111` which holds 39 different tokens or `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead` which holds 21 different tokens. One exception is the zero address where people have unintentionally locked tokens. Indeed, Solidity automatically pads values with zeroes. As a result, calling a function such as ERC20's `transfer` with a `to` parameter uninitialized will result in a transfer of tokens to the zero address. This is part of the reason which led to the decision to forbid sending tokens to the zero address address in ERC777.

5.2.2 Location Of The Hooks

One essential aspect is where those hooks are located. One approach is to have those hook functions located at the recipient, but this has two significant drawbacks. First the recipient must then be a contract to implement the hooks—hence regular accounts could not use hooks. Secondly, existing contracts do not implements the hooks and could not receive ERC777 tokens.

The approach used in ERC777 is to use a registry to lookup the address of the contract implementing the hook for a given recipient. This approach has many advantages over the previously mentioned one. Primarily, all addresses, even regular accounts, can use the registry to register a contract implementing the hook on their behalf. Second, this means that existing contracts can also register hooks via a proxy contract which implements the hook on their behalf. Essentially this means that an account or an already deployed contract can simply deploy a new contract to implement these hooks on their behalf.

ERC777 relies upon this registry, but the registry itself is not part of the standard. Instead, the registry is specified in a separate standard, ERC820: A Pseudo-introspection Registry Contract [Baylina and Dafflon, 2018], outlined in chapter 6. ERC777 then simply relies upon ERC820. The advantage of dissociating the token standard from the registry is that first it can be used by other standards and secondly it offers a good separation of concerns. Any developer wishing to work with ERC777—whether it is to implement a token or any kind of DApp—will need to thoroughly understand ERC777 in order to deploy code which is compliant. In comparison, the ERC820 registry should already be deployed and the developer only need to understand how to properly interact with it.

5.3 Sending Tokens

Unlike ERC20 which defines a couple functions to send tokens, ERC777 focuses on specifying the process which must be followed when sending tokens. The standard then enforces the presence of two functions—`send` and `operatorSend`—which apply the send process. Other non-standard functions may be added when creating an ERC777 token contract, as long as those functions follow the specification of the send process.

The send process works as follow. First only an authorized operator can send the tokens of a token holder. This includes the token holder itself, a non-revoked default operator (if any) or some other explicitly authorized operator. Second, a few (obvious) rules must be enforced, such as the recipient cannot be the zero address. The amount to send must not be greater than the balance of the token holder, the amount must be a multiple of the granularity (see **??**) and the appropriate balances updated accordingly, a `Sent` event must be emitted and more importantly, the `tokensToSend` and `tokensReceived` hooks must be fired before and after updating the balances, respectively.

Another aspect of sending tokens is the `data` and `operatorData` parameters, which is an undeniable improvement over ERC20 where only the recipient and amounts of tokens can be specified as part of a transfer. The `data` parameter is intended to be similar to the `data` parameter of a regular Ethereum transaction and the standard intentionally does not enforce a specific format for this data only that the recipient defines what data it expects. We expect people will propose new standard related to the format of the `data` parameter which define the format required for a specific use case. ERC1111[Drake, 2018] is an effort in this direction.

5.4 Minting Tokens

5.5 Burning Token

5.6 View Functions

The view functions in ERC777 have been taken from ERC20. Since those functions do not modify the state, they can be used interchangeably for both standards without creating conflicts. The only constraint is that the information returned by these functions must represent the same entity.

Specifically the `name`, `symbol`, `totalSupply` and `balanceOf` functions are kept. The differences with ERC20 is that the cosmetic `name` and `symbol` function are now mandatory. In addition there are more strict constraints on the value returned by calls to the `totalSupply` and `balanceOf` functions.

5.7 Decimals

The ERC20 `decimal` function is notoriously absent from the view functions listed above. As previously explained, a variable `decimals` value is problematic. For this reason the `decimals` has been set at a fixed value of 18. This renders the `decimals` function pointless. The standard only enforces the implementation of the `decimals` function when implementing a ERC20 backward-compatible token. In this case the `decimals` function must both be implemented and return 18. The choice has been made to make the `decimal` function mandatory in this case, even though ERC20 considers the function optional. The rational behind this decision comes from the lack of an explicit value defined in the ERC20 standard when the `decimals` function is not defined. Furthermore, requiring people to check wether a token is both ERC20 and ERC777 compatible—and then deduct from the ERC777 standard that the number of decimals is 18—is both unreliable and terrible user experience. Besides, this would add an opaque constraint when implementing both standards.

5.8 Compatibility

One key aspect for the ERC777 standard is to maintain compatibility with the older ERC20 standard. The Ethereum ecosystem is hard and slow to update. This translates to many wallets, DEXs and other DApps which today support ERC20 but will not support ERC777 for years to come if not ever. Hence ERC777 tokens will not be supported on existing platforms immediately, which is a problem for people wishing to sell and trade their ERC777 token. Having a token able to behave at first like an ERC20 token on those platforms alongside with the newer ERC777 behavior is a major social and economic advantage.

5.8.1 Notifying The Recipient

One of the main feature of ERC777 is to do away with the `approve` and `transferFrom` mechanism and instead provide a single mechanism to send tokens to both regular accounts and contracts. For a contract, it is then mandatory for it to be notified on-chain when it receives tokens. This is done with the use of “hooks” or specific functions which are called

5.8.2 ERC20 Backward Compatibility

Many proposals try to fix ERC20 and amend the specifications to modify the behavior of functions such as `transfer`. A new standard on the other hand should not modify the behavior of existing functions defined in another standard, but rather define different functions which implement the new behaviors.

Additionally, having functions with disparate names allows people to differentiate with which standard they are interacting with. As an example, with ERC20, to send tokens to a contract, a user should typically use `approve`—instead of calling `transfer` directly—and let the recipient call `transferFrom`. On the other hand, ERC777 defines a `send` function (described later) which is safe to use to send tokens to a contract. Let us assume this `send` function was named `transfer` instead. The user must know figure out beforehand if the tokens he wishes to send implement ERC20 or ERC777. This burden also applies to contract forwarding contracts. Overall having the same name would create confusion and result in many mistakes where people end up loosing their tokens.

Explain ERC777 in thorough details:

- specifications
- ERC820
- `TokensSender` and `TokensRecipient`
- Operators
- ERC20 compatibility
- Collaboration with Jordi Baylina
- Public Reception

Chapter 6

ERC820, Pseudo-introspection Registry contract

Chapter 7

Competing Token Standards

7.1 ERC227

7.2 ERC827

7.3 ERC995

Chapter 8

The State Of Tooling

8.1 Compilation

8.2 Testing and Coverage

8.3 Documentation

8.4 Missing Tools

Chapter 9

Future Research and Work

- ERC777
  - Generic operators
  - Generic TokensSender And TokensRecipient
  - Promote ERC777
  - Assist In The Implementation of ERC777 Technologies (Wallet, Exchanges, Blockchain Explorers)
- Payment Channel
  - Other use cases for tokens (actually voting with your wallet)
  - Tax deductions on some tx (add tag in payment channel?)
- Loyalty Programs
  - On chain (ETH rewards, tokens Rewards, Pay with Tokens)
  - Off chain (T-shirts, coffee machines and toasters)

Chapter 10

Conclusion

I'l do this one at the end

Glossary

ABI

Application Binary Interface. 14, 33

DApp

Decentralized Application. 8, 9, 20, 21, 33

DEX

Decentralized Exchange. 18, 21, 33

EVM

Ethereum Virtual Machine. 5, 33

LIFO

Last-In-First-Out. 5, 33

zero address

The zero address in Ethereum is the address composed only of zeroes, i.e. `0x0000000000000000000000000000000000000000`. It is commonly used to represent the minting or burning of tokens with ERC20. It is also used as the destination address of a transaction deploying a contract. 19, 20, 33

Bibliography

Jordi Baylina and Jacques Dafflon. ERC820: Pseudo-introspection registry contract, January 2018. URL https://eips.ethereum.org/EIPS/eip-820.

Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2013. URL https://github.com/ethereum/wiki/wiki/White-Paper.

Jacques Dafflon, Jordi Baylina, and Thomas Shababi. ERC777: A new advanced token standard, November 2017. URL https://eips.ethereum.org/EIPS/eip-777.

Chris Drake. ERC1111: Introspective metadata definition disclosure (discussion to establish a standard), May 2018. URL https://github.com/ethereum/EIPs/issues/1111.

Ethereum Foundation. EIP X, the suggested template for new eips, 2015. URL https://github.com/ethereum/EIPs/blob/master/eip-X.md.

Ethereum Foundation. Ethereum improvement proposals, March 2018a. URL https://eips.ethereum.org/.

Ethereum Foundation. Solidity 0.4.24 documentation, May 2018b. URL https://solidity.readthedocs.io/en/v0.4.24.

Mikhail Vladimirov and Dmitry Khovratovich. ERC20 API: An attack vector on approve/transferFrom methods, November 2016. URL https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM.

Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, November 2015. URL https://eips.ethereum.org/EIPS/eip-20.

Micah Zoltu, August 2018. URL https://github.com/ethereum/EIPs/issues/1298#issuecomment-410966921.