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 xi

List of Tables xiii

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 . . . 21
5.6 Data And Operator Data . . . 21
5.7 View Functions . . . 22
5.8 Decimals . . . 22
5.9 Compatibility . . . 22
5.9.1 ERC20 Backward Compatibility . . . 22
5.10 Community And Public Reception . . . 23
5.10.1 Jordi Baylina . . . 25

6 ERC820: Pseudo-introspection Registry Contract 27
6.1 First Attempt, The ERC165: Standard Interface Detection . . . 27
6.2 Second Attempt, The ERC672: ReverseENS Pseudo-Introspection, or standard interface detection . . . 28
6.3 Final Attempt, The Need For The ERC820 Registry . . . 28
6.4 Separation Of Concerns . . . 28
6.5 ERC165 Compatibility . . . 29
6.5.1 Caching ERC165 Interfaces . . . 29
6.6 Registry Interface . . . 29
6.6.1 Interface Identifier . . . 29
6.6.2 Lookup . . . 30
6.6.3 Setting An Interface . . . 31
6.6.4 `setManager` . . . 31
6.6.5 `getManager` . . . 32
6.6.6 `interfaceHash` . . . 32
6.6.7 ERC165-specific functions . . . 32
6.7 Keyless Deployment And Unique Contract Address Across All Chains . . . 33
6.7.1 Vanity Address . . . 35

7 Competing Token Standards 41
7.1 ERC227 . . . 41
7.2 ERC827 . . . 41
7.3 ERC995 . . . 41

8 The State Of Tooling 43
8.1 Compilation . . . 43
8.2 Testing and Coverage . . . 43
8.3 Documentation . . . 43
8.4 Missing Tools . . . 43

9 Future Research and Work **45**

10 Conclusion **47**

Glossary **49**

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 and relying upon the state of this Virtual Machine rather than dealing with transaction outputs; transactions simply modify the state.

This idea of a global computer allows one to write a program, hereinafter a Smart Contract, which interacts with the EVM and inherits the safety properties of the Ethereum system (and also its limitations). Essentially it is a very low power/capacity computing platform with interesting safety properties (such as operations and state data being essentially immutable once a transaction is included in the blockchain [with sufficient confirmations as its probabilistic after all]. This is ideally suited to small minimalistic programs governing essential data, such as a ledger of transactions.

One such example of smart contracts is the ERC20 token standard (there are varying smart contract implementations). This is likely the most widely deployed smart contract on Ethereum. One issue is the design of ERC20. 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. This also limits the way smart contracts can interact with ERC20 tokens and adds complexity to the User eXperience (UX).

The new ERC777 token standard solves these problems and offers new powerful features which facilitate 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., 2017a] 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 (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.

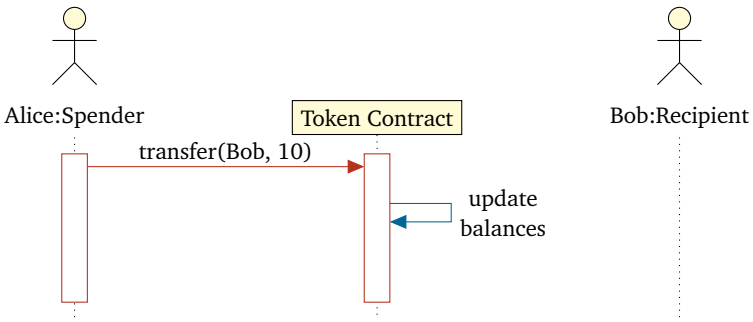

Figure 4.1. Standad ERC20 `transfer` between two regular accounts: Alice and Bob.

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.

```
1 /**
```

```
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.

```
1  modifier isRunning {
2      assert (!stopped);
3      _;
4  }
5
6  modifier validAddress {
7      assert(0x0 != msg.sender);
8      _;
9  }
10
11 // ...
12
13 function transfer(address _to, uint256 _value)
14     isRunning validAddress returns (bool success)
```

```
15 {
16     require(balanceOf[msg.sender] >= _value);
17     require(balanceOf[_to] + _value >= balanceOf[_to]);
18     balanceOf[msg.sender] -= _value;
19     balanceOf[_to] += _value;
20     Transfer(msg.sender, _to, _value);
21     return true;
22 }
```

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.

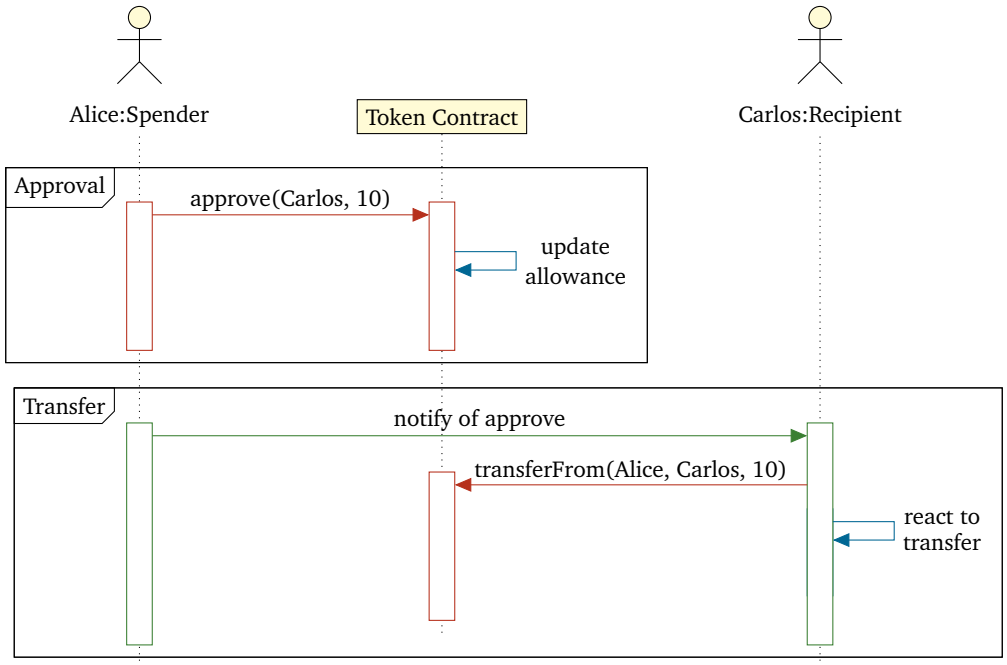

Figure 4.2. Standad ERC20 `transferFrom` between a regular account Alice and a contract account Carlos.

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 UX 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.

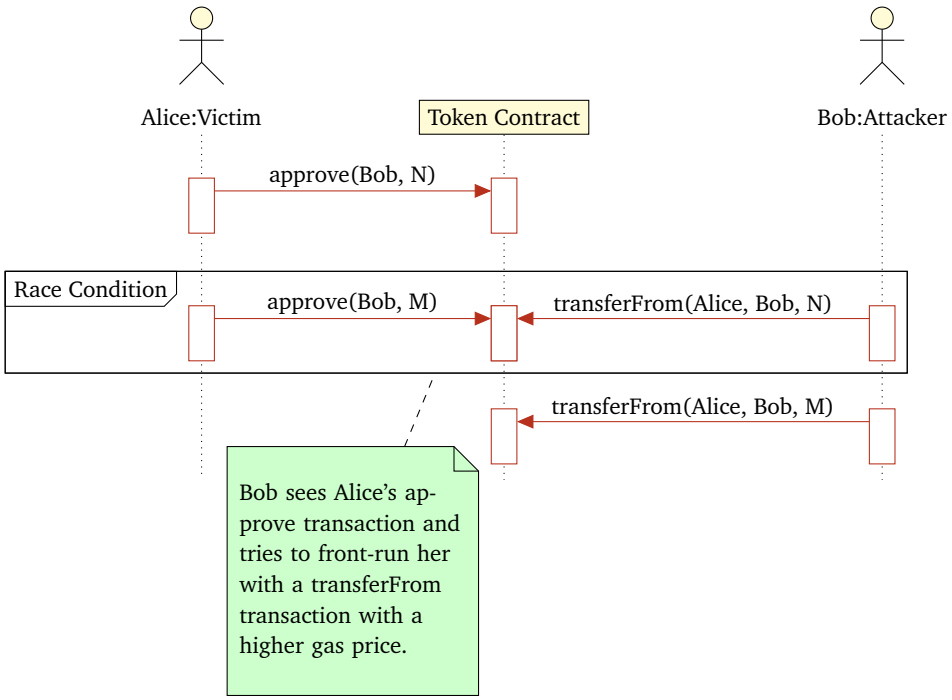

Figure 4.3. Sequence of calls for the `approve`/`transferFrom` attack.

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 [Buterin, 2017] 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., 2017a].

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 the transaction 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.

5.4 Minting Tokens

Minting is the technical term referring to the creation of new tokens—it originates from the minting of metal coins. The creation of tokens in Ethereum is extremely specific to the asset represented by the token and involves various mechanisms accordingly. Some tokens have a fixed amount of tokens minted at creation time—often referred to as initial supply—which is given to the user(s) controlling the contract. Other tokens have an issuance model which mint tokens according to signed messages provided by a trusted third party.

On one hand, because of these varying issuance models, it is hard to provide a standardized process which creates tokens therefore this is intentionally left out of ERC777. On the other hand, ERC777 does define a set of rules which must be respected when minting new tokens. These rules include:

1. The total supply must be updated to reflect the mint.
2. The tokens must be minted for an account whose balance must be increased
3. A `Minted` event must be fired.
4. The `tokensReceived` hook must be called if present.
5. If the recipient is a contract which does not have a `tokensReceived` hook, the minting process must revert.

From the recipient point of view, minting and sending tokens is similar. The main difference is with minting the `from` address is the zero address which indicates the tokens are newly created. The notion of operator is also slightly different for minting. As mentioned in chapter 5.1, an operator is an address which can spend the tokens of some account (either through sending or burning). This notion does not apply to minting as no one previously owns the minted tokens. ERC777 does not enforce any constraint on which address can mint tokens. It is up to each token to define condition in order to restrain the minting process such that it matches the desired issuance model. For example, the minting can be fully restricted, only allow some addresses to mint, or only allow minting in certain quantities, at certain times or if some other condition is met such as providing a signed message. These various issuances model are the reason why there is no explicit function for minting as part of the standard.

5.5 Burning Token

Burning tokens, similarly to minting can be specific to which asset a token represents. Some token contracts may wish to never allow burning of token, Others may only allow some addresses to burn token, and some may allow anyone to burn tokens if specific conditions are met. Lastly, token contracts may want or need to take specific actions when tokens are burned, e.g., a token may represent a redeemable asset where the token is burned in order to redeem the asset.

Similarly to minting, burning applies rules identical to send, but in this instance on the token holder or spender. I.e. equivalently to a regular send, an operator must be authorized to burn the tokens and the `tokensToSend` hook of the token holder must be called, the only difference compared to a send is that the recipient—the `to` parameter—of the hook is set to the zero address when burning. Note that when burning the actual balance of the zero address must not be increased. As a side note, the constraint coupled with the constraint that sending to the zero address is forbidden, implies that it is impossible for the zero address to ever hold any ERC777 token.

5.6 Data And Operator Data

Another aspect of minting, sending, and burning 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.

The `operatorData` is the second free bytes parameter associated with a token transaction. This parameter is explicitly intended for the operator to provide any information or reason for the transaction. In contrast, the `data` may be provided by either the token holder, the operator, or both. Ultimately the recipient is free to define which `data` it expect and reject any transaction which does not matches its expectations.

5.7 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 or metadata `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. For example, the value returned by the total supply must be equal to the difference between the sum of all the `Minted` events and the sum of all the `Burned` events.

5.8 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 UX. Besides, this would add an opaque constraint when implementing both standards.

5.9 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.

5.9.1 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.

Fundamentally, ERC777 allows for a token to be implemented as both an ERC20 token and a ERC777 token. This hybrid token possess two distinct behaviors—one per standard—and the choice of which behavior is considered is left to anyone who interacts with the token. On one hand, if a user issues an ERC20 `transfer` call, then for him the token behaves as an ERC20 token. When the user issues an ERC777 `send`, the token behaves as an ERC777 token for him. On the other hand, if the recipient expects to receive ERC777 tokens, he will see a reception as an ERC777 reception, regardless of which function the operator used to send the tokens. Likewise if the recipient expects an ERC20 token he can see the reception of tokens as an ERC20 transfer. Even for third party who observe a token contract, they can choose to observe the token as an ERC20 token and listen to `Transfer` events or as an ERC777 token and listen to `Minted`, `Sent`, and `Burned` events.

This behavior is achieved by enforcing that for any transfer of tokens (using either ERC20 or ERC777), both a `Sent` event and a `Transfer` event must be emitted. Correspondingly for minting and burning, along side the ERC777 `Minted` and `Burned` events, an ERC20 `Transfer` event with `from` and the `to` field set to the zero address respectively. This is effectively a stricter constraint than ERC20 which only recommends—but does not require—a `Transfer` event with the `from` field set to the zero address and does not specify the concept of burning. The reason for this stricter constraint is to maintain consistency across the standards and to provide the same data regardless of which standard is used.

It should be noted that defining ERC20-related constraints in ERC777 does not conflict with ERC20. Adding the constraints to ERC20 directly is problematic as it would make existing tokens non-compliant, although it is not an issue of the constraint is expressed in ERC777 and they only apply to ERC20–ERC777 hybrid tokens and none of them exist to this date. ERC20 was intentionally defined more loosely to ensure that it would make some existing tokens retroactively compliant. With the new process for EIPs, we have the opportunity with ERC777 to clearly state that the standard is still in a draft phase and should not be used. This of course does not prevent people from trying to implement the standard, however breaking changes may still happen at this stage and it is up to the token designer to make sure their implementation is compliant with the final version of ERC777 once it is finalized. Hence we are do not have to worry about having to weaken the standard to support some existing and poorly-implemented token. Efforts will need to be put into ensuring the first developers properly implement the standard and we have already personally and privately contacted the chief technical officers or founders of some start-up to inform them that their current implementation is not compliant with the latest version of ERC777.

5.10 Community And Public Reception

An important factor towards the finalization and the success of this standard is how well it is received by the community. It was crucial to remain open and listen to the views, suggestions

and feedback from the community. Most of the feedback has been provided publicly via comments on the ER777 issue [Dafflon et al., 2017b], some feedback was also given privately via email, instant messages or in person–mostly when meeting other developers at conferences and events.

When reading any comments, instead of going away with a fixed mindset and standardizing our own view, we adapted the standard to accommodate for the feedback of the community. Obviously such effort requires some filtering as not every comment can result in a change of the standard. Some of the messages were inaccurate or wrong due to misunderstanding of the standard or lack of knowledge regarding the Ethereum ecosystem. In such situation, it was important for us not to ignore those comments but to reply and try to explain or clarify the topics which misinterpreted. Doing so gave us the opportunity to understand where the inaccuracies came from and clarify the standard to provide an explicitly and clear message for all future readers. Some of the readers or developers who will use the standard may not be native or even proficient English speakers and it is paramount to make the text plain enough to be understood by all and accessible to to anyone.

Some of the comments have provided valuable information which resulted in changes to the standard. An example include how `decimals` and `granularity` is handled. Initially the `decimals` function was part of ERC777 and similar to ERC20. Today the function has been removed from the standard, the number of decimals is fixed to 18 and the concept of granularity and the `granularity` has been defined.

Another example is the `tokensToSend` hook. At the beginning the standard had a single hook named `tokenFallback`. This hook was later renamed to `tokensReceived` and the new `tokensToSend` hook was added based on a suggestion of a community member and a general agreement from multiple people that this hook was useful. The addition of the `tokensToSend` hook was not a straightforward decision as some people—including myself—initially disagreed with the `tokensToSend` hook. The issue number 23 of the ERC777 reference implementation [Dafflon, 2018a] contains a detailed explanation of the drawbacks of the `tokensToSend` hook [Dafflon, 2018b]. No solution is perfect or optimal and this hook does have some drawbacks. Notably it adds some complexity to the logic of the token contract and the movement of tokens. The gas cost of a send becomes more variable as the absence or existence of the hook and the various implementations add more entropy to the actual code execution performed when sending tokens. Another aspect is some of the checks potentially implemented in a `tokensToSend` hook may be implemented via an operator instead.

Ultimately, the complexity increase is less than the advantages brought by the hook, the gas cost can already be quite variable based on `tokensReceived` hooks and while some functionalities can be implemented either in an operator or a `tokensToSend` hook, there is a significant difference: the functionality located within the operator only applies to this specific operator and can be bypassed by using another operator. It also requires said operator to be a contract. Deploying a `tokensToSend` hook for the functionality allows the logic to be applied to all debit of tokens regardless of which operator it originates.

Unfortunately, some of the comments—originating from what is commonly referred as a internet troll—are clearly negative and do not bring any constructive criticism. Those comments can be frustrating and diverge the discussion from the actual work into the emotional realm. Thankfully there was never any extreme case so far with respect to the development of the standard. The conventional internet wisdom: “Don’t feed the trolls” worked in our case and the members of community were all wise enough to not take the bait. The best solution was to approach the topic with humor and redirect it away from the EIP discussion.

5.10.1 Jordi Baylina

Jordi Baylina is a member of the White Hat Group and Giveth and one of the authors of the ERC777 standard. Being a well known and well appreciated figure in the Ethereum community, made it easier to promote ERC777 and convince people of the quality and seriousness of the standard. Thanks to his many years of experience in Ethereum both as an end-user and as developer, he is able to provide valuable input on the design of the standard both from a security and a usability point of view. He is also a great source of knowledge regarding some of the quirks of Ethereum and Solidity which was essential to help me gain personally a deeper understanding of the Ethereum ecosystem.

Chapter 6

ERC820: Pseudo-introspection Registry Contract

ERC820 is the standard defining the registry used by ERC777 to let addresses—regular accounts and contracts—register the address of a contract implementing the required interface for them. Within the context of ERC777, any address may register a `TokensSender` interface and a `TokensRecipient` interface which are the interfaces for the `tokensToSend` hook and the `tokensReceived` hook respectively. This allows the token contract to know where the implementation of the hook is located and to execute it.

In addition the token contract itself must register its own address as implementing the `ERC777Token` interface which corresponds to the interface any ERC777-compliant token contract must implement. In addition if the ERC777 token is actually an ERC20-ERC777 hybrid token, it must register the `ERC20Token` interface as well, and if the token contract has a switch to disable ERC20 or ERC777 functions, then the contract must dynamically register and unregister its address for the corresponding interfaces.

6.1 First Attempt, The ERC165: Standard Interface Detection

ERC165 was created on January 23rd 2018 and finalized on February 21st the same year. It is a simple and short specification which allows to interact with a contract directly to detect if the contract implements a specific function. While this standard could be used for ERC777 to detect if a recipient contract implements the `tokensReceived` hook, it is very limited in that only contracts and not regular accounts can use the hook and it does not allow contracts to delegate the implementation of the hook to a proxy contract.

This standard has significant drawbacks which as it is, would automatically make ERC777 incompatible with all existing contracts, including multsig wallet which can hold large sums of ether and tokens and whose migration to a new contract is both is a sensible subject both from a security and a safety point of view if people are not careful. Hence it was decided a better alternative should be used.

6.2 Second Attempt, The ERC672: ReverseENS Pseudo-Introspection, or standard interface detection

ERC 672: ReverseENS Pseudo-Introspection, or standard interface detection [Baylina and Izquierdo, 2017] was the second attempt at creating a better solution which could fulfill the primary motivation behind ERC777: Designing a system—such as a registry—that given a contract recipient, the token contract would be able to find the address of some contract—the recipient or other—which implements a function with the logic to notify the recipient contract such that the tokens are not locked.

This first attempt relied on ENS and implementing a reverse ENS lookup through a registry contract. Overtime however we came to realize this attempt may be overly complicated unsuitable for security reasons. Indeed, this solution relies on ENS and interactions with ENS complicate the task of resolving the interface. Furthermore, ENS is still controlled by a multi-signature contract and theoretically with enough of the keys the system could be corrupted.

6.3 Final Attempt, The Need For The ERC820 Registry

At this point, the need for a separate independent registry became clear. This is where from ERC820 was born. From the ERC820 standard itself:

> [It] defines a universal registry smart contract where any address (contract or regular account) can register which interface it supports and which smart contract is responsible for its implementation [Baylina and Dafflon, 2018].

This solution offers solves the issues of the attempts from ERC165 and ERC672. Namely, it can be used by both contracts and regular accounts, it does not rely on ENS and therefore it is much simpler and does not inherit any of the trust or security concerns from ENS.

6.4 Separation Of Concerns

The ERC777 standard relies on the ERC820 registry to work as intended. Without the registry, it is not possible to move tokens in a compliant way. A fair proposal would be to only submit a single standard containing the specification for tokens and the registry. However while developers will have to implement token contracts, no developer is expected to implement the registry, thus moving the registry in its own standard is a good separation of concerns. At most they may use the provided raw transaction and broadcast it on the chain they use if the registry is not already deployed. Furthermore, the ERC820 registry may actually be used independently of ERC777. Other standard or simply DApps may use it lookup implementers of specific interfaces they need. Splitting the standard in this way gives us the opportunity to make available some of the more generic work needed for ERC777 for other tasks which are not ERC777—or even tokens—specific.

6.5 ERC165 Compatibility

Furthermore, the ERC820 registry is compatible with ERC165 and is able to act as a cache for ERC165, thus saving gas when querying for an interface. More than just saving gas, one can query the ERC820 registry to find out if a contract implements an interface and use dynamic values for the interface to query without having to worry or check if the interface is an ERC165 interface or an ERC820 interface.

6.5.1 Caching ERC165 Interfaces

Caching with respect to ERC165 is rather simple. Since the code of a contract is immutable, once a contract is deployed with a given interface it cannot change the interface over time. The only times the interface can change is when the contract is created and when the contract is destroyed. In those cases, the cache needs to be manually updated, as there is no automatic cache invalidation or cache update process. This is an explicit choice to keep the registry simple and keep the gas consumption low. The section 6.6.7 describes the function needed to update the cache.

6.6 Registry Interface

The registry exposes five main functions as part of its interface with an additional three functions specific to ERC165. These three functions are used internally as part of the main functions but can also be used directly if needed.

6.6.1 Interface Identifier

Each interface has a unique identifier, which is the `keccak256` hash of the interface name. This name is just an arbitrary name decided by the implementer of the interface. The ERC820 standard does defines some rules for interface names. First the name of interfaces for a specific Ethereum Request for Comments (ERC) must be of the form `ERC###XXXXX` where `###` is the ERC number and `XXXXX` is the actual name for the interface. As an example, for an ERC777 token the interface name is `ERC777Token` which indicates this is part of the ERC777 standard and it is the `Token` interface. Alike, The `ERC777TokensRecipient` is the `TokensRecipient` interface for ERC777.

Other interfaces for private use are free to define their interface name freely but those name must not conflict with names for ERCs. One recommendation is to prefix the interface name with the name of the company or product related to the interface. Ultimately, there is no central authority or enforcement of any kind—externally or internally in the code of the registry—to ensure those rules are respected. Developers should follow the standard and if they deviate from it and do not respect it, they run the risk of their DApp not working or suddenly breaking down.

Since all interfaces are hashed using `keccak256` they are all 256 bits, i.e thirty-two bytes long, with the exception of ERC165 interfaces which are only four bytes long. This difference of hash length is the property used to detected whether an interface is related to ERC165 or ERC820. Specifically, all functions expects thirty-two bytes parameters—i.e parameters of type `bytes32`. Since Solidity pads parameters with zeroes, a four bytes parameters is automatically

padded to thirty-two bytes with twenty-eight zeroes and those zeroes are used to detect if the interface is related to ERC165. The listing 6.1 shows the implantation for this logic in the ERC820 registry.

```
1 /// @notice Checks whether the hash is a ERC165 interface (ending with 28
      zeroes) or not.
2 /// @param _interfaceHash The hash to check.
3 /// @return `true` if the hash is a ERC165 interface (ending with 28 zeroes),
       `false` otherwise.
4 function isERC165Interface(bytes32 _interfaceHash) internal pure returns (
      bool) {
5     return _interfaceHash & 0
          x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF ==
          0;
6 }
```

Listing 6.1. The logic use to detect if an interface is related to ERC820 or ERC165 in the ERC820 registry.

6.6.2 Lookup

The `getInterfaceImplementer` is the function used to lookup the address of the contract implementing a given interface for a specific address. It takes as first parameter the address for which to lookup the implementation and as second parameter the hash of the interface name to be queried.

If the interface is four bytes long and part of ERC165, then the registry will first look in its cache and then directly with the specified address—using ERC165—to lookup if the address implements the interface. If it does, then the address itself is returned otherwise the zero address is returned. With ERC165, since it does not support the concept of proxy contract, only the address passed as parameter or the zero address can be returned and if the address passed is not a contract then the returned value is always the zero address.

Furthermore in this scenario, if the lookup is called as part of a transaction and the cache is uninitialized, then the cache will automatically be updated. This implies a slight increase in gas cost for the first transaction, however future transaction will consume less gas as they will rely on the cache.

If the interface is a full thirty-two bytes long, then the function will return the address from an internal double mapping. This double mapping relies on the internal `mapping` type of Solidity which returns the default value for the type of the mapping value if the key is not present. As the mapping value is of type `address` it returns the zero address by default if there is no entry in the double mapping for a given address and interface hash. If a specific address is present however then this address is returned. Overall this means that if the returned address is the zero address then the given address does not have a contract implementing the interface, if the address is different then it is the address implementing the interface for the given address. If the given address and the address implementing the interface are the same, then it means the given address which implements the interface for itself.

6.6.3 Setting An Interface

The `setInterfaceImplementer` function is used to set the address of the contract implementing the given interface for the given address. For obvious security reasons, not every address is allowed to set an interface implementation for a given address. Only the manager of an address is allowed to set the implementation of an interface for the given address. By default every address is it's own manager but each address can set another address as its manager using the function described in section 6.6.4.

Furthermore to avoid addresses settings random contracts as interface implementers for themselves, if the address for which to set the implementation and the address of the implementer differ, then the ERC820 registry requires for the implementer to implement the `ERC820ImplementerInterface` interface which consist of a single function: `canImplementInterfaceForAddress` detailed in section **??** below.

The `canImplementInterfaceForAddress` Function And The "Accept Magic" Return Value

This function must be implemented by the implementer of some (other) interface. The function gives an opportunity for the implementer not to be set as the implementer for an address it does not expect to be the implementer of the given interface for that other address. As an example, if a user creates an implementation of the `ERC777TokensReceived` for its specific address and the implementation is not able to handle the `ERC777TokensReceived` for more than the expected address of the user, then the `canImplementInterfaceForAddress` must only return the "accept magic" value for the expected address.

This "accept magic" value is actually the `keccak256` hash of the string `ERC820_ACCEPT_MAGIC` which is `0xf2294ee098a1b324b4642584abe5e09f1da5661c8f789f3ce463b4645bd10aef`. This value is considered as `true` and any other value is considered as `false`. This may appear as a strange way of approaching the problem and that a boolean—which is supported by Solidity—may be a much better approach, albeit the reason for this approach is to avoid a Solidity quirk which might return a false positive when calling `canImplementInterfaceForAddress` on the implementer. In the case where the implementer contract fails to implement the `ERC820ImplementerInterface` and the `canImplementInterfaceForAddress` function but implements a fallback function which does not throw, then in this case the fallback function will be called instead of the lacking `canImplementInterfaceForAddress` function and the fallback function will return `1` which is coerced to `true`.

The registry of course does not allow settings an ERC165 interface, this is done entirely on the contract itself by following ERC165.

6.6.4 `setManager`

By default, every address is its own manager which means it is the only address allowed to set the implementer of an interface for itself. This role can be transferred to another address—there can only be one manager for any address at any given moment in time. The `setManager`—obviously—can only be called by the manager itself. Therefore only the manager can transfer its role to another address. Furthermore if the current manager sets the zero address address, this is interpreting as reseting the address itself as its own manager. If an address does not want to have a manager, then it can always set the manager to some address made of a "deterministic"

pattern such as `0x1111111111111111111111111111111111111111` or `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead`.

(address _addr, address _newManager) external {

6.6.5 `getManager`

The function is a simple getter to get the address of the manager for a given address. While the address is its own manager by default, the address of the manager can be updated to another address. Hence this functions allows to obtain the actual address of the manager regardless of whether the address of the manager has been changed or not.

6.6.6 `interfaceHash`

The `interfaceHash` is merely provided as a convenience. It returns the `keccak256` hash of the interface name which is passed as a string. Ideally, the hash should be computed off-chain and passed directly to the registry, however in the case where this might not be possible, the `interfaceHash` is available to hash the string and return the correct hash.

6.6.7 ERC165-specific functions

The following functions are specific to ERC165 and are used internally by the `getInterfaceImplementer` function if the interface hash is a four bytes ERC165 interface hash to figure out if the given contract does implement the interface.

`implementsERC165Interface`

The `implementsERC165Interface` function is the function called by `getInterfaceImplementer` function for ERC165 to find out if a contract implements the given ERC165 interface. It uses the internal cache of the ERC820 library and only actually check if the interface directly with the contract if the cache is not set.

`implementsERC165InterfaceNoCache`

The `implementsERC165InterfaceNoCache` function should returns the same result as the `implementsERC165Interface` function but internally the formed does not use the cache at all and is actually used by the latter to set the value in the cache if it is not set.

`updateERC165Cache`

The `updateERC165Cache` is called within the `implementsERC165Interface` to set the cache value if it is not set. Internally this function uses the value returned by the `implementsERC165InterfaceNoCache` function.

Moreover the `updateERC165Cache` can be called directly> For example if the contract was not deployed and the registry cached (rightfully at the time) that the address of the contract did not implement the interface, this function can be called after the deployment of the contract to update the cache to reflect that the contract is now deployed and implements the interface. The same applies the other way around, if a contract self destructs, then this function can be called to reflect that this address does not implement any interface anymore.

6.7 Keyless Deployment And Unique Contract Address Across All Chains

It is paramount for the ERC820 registry not to be controlled by anyone. If any account has control over the registry, this account may manipulate or potentially destroy the registry. Hence the registry must be deployed from an address from which not only no one controls the private key but everyone must also easily be convinced of the fact that no one controls the private key.

There is nice—and somewhat unknown—feature of Ethereum which we can take advantage of to achieve this goal: keyless deployment using a single-use Ethereum address for which no one has the key. This method is also referred to as "Nick's method" as an acknowledgment to Nick Johnson who suggested this method for ERC820.

In order to understand how this method works, one must first comprehend how a transaction is signed in Ethereum and how the address of the sender—which is not explicitly part of the transaction—is recovered. In Ethereum, the transactions are signed using Elliptic Curve Digital Signing Algorithm (ECDSA). To send a verified transaction, one must generate a message and sign it using their private key. This signed message is the authorization to spend a specific amount of ethers from the account. Precisely, this signed message is made up of the following components forming an Ethereum transaction: the `to` value (i.e. the recipient), the `value` (i.e. the amount of weis to spend), the `gas` (i.e. the gas limit or the maximum amount of gas the transaction is allowed to spend), the `gasPrice`, (i.e the price of each unit of gas in weis), a nonce and the `data` field. The signing number returns an Ethereum signature composed of three numbers, commonly referred to as `r`, `s`, `v`. The numbers r and s are defined by the ECDSA algorithm and define the coordinate on the curve—extremely roughly r is the x-value and s is the y-value of the coordinate.

The value v is defined in the Ethereum Yellow Paper as $v \in [27, 28]$, more precisely:

> It is assumed that v is the 'recovery identifier'. The recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value; this value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid. The value 27 represents an even y value and 28 represents an odd y value [Wood, 2018, Appendix F].

Ethereum defines a function knows as `ecrecover` which given the message hash and the three numbers `r`, `s` and `v` is able to recover the public key and thus the address of the spender which signed the transaction. Obviously because only the corresponding private key could generate valid values for `r`, `s` and `v` such that it results in the correct public key and therefore the correct address.

Single use functions come from the answer to a simple question: What if someone generate a valid transaction such as a signed message to send ether to a specific address and then use some random values for `r`, `s` and `v` which are hard coded and not derived from some private key? Now the hash of the message and the `r`, `s` and `v` values can be paased to `ecrecover` to obtain the origin address for this transaction. Moreover, the transaction can be broadcasted on the Ethereum network and if the origin address has the funds they will be transferred! Thus we have just achieved a transfer of ethers from an address for which we do not know the private key.

Before being thrown into widespread panic that your funds are insecure and may be spent by anyone able to craft a transaction, it is very important to note that this method does not provide any control to select the origin address for the transaction. The origin address is derived using ECDSA which is cryptographically secure and generating a transaction this way—without knowing the private key—for a specific origin address would require to brute-force multiple values for `r` and `s` until values which derive to the desired address are found. (`v` is defined as $v \in [27, 28]$, hence it is trivial to cover this key space.) This is equivalent to brute-forcing the private key and then using it to generate the correct `r` and `s` values, and brute-forcing the private key is today computationally infeasible.

Nonetheless, this process of generating transaction is useful for single-use address. Essentially it is computationally infeasible and probabilistically extremely unlikely that a second transaction for the same address can be generated. But we manage to generate a single transaction for this address and if we send enough ether to this address (including ether to pay for the gas) before broadcasting our transaction, once the transaction is broadcast the ether from that address will be spent and credited to the address we set as recipient in the transaction.

Nick Johnson describes this method and provides an example on how this method can be taken advantage of in a nested or recursive fashion to send ether to a large amount of addresses from a multsig wallet [Johnson, 2016] while only signing a single message containing a series of transaction—and this can be done regardless of the number of final recipients. Essentially a series of transaction are generated, one per final recipient and they are then put in batch into different signed messages which are signed. The origin addresses for those signed messages are then derived and the process repeats, generating transactions to send ether to those derived addresses which are batched and so on until there is only a single batch in a single signed message and thus a single address to derive.

TODO add picture

Next this tree of transactions is shared off-chain with the owners of the multsig wallet wallet and they can inspect all the transactions, then once convinced they only have to approve a single transaction from their multsig wallet to the origin address of the root of the transactions and then broadcast each level of the transaction in order to have the ether sent to all the recipients automatically.

In the case of the ERC820 registry we do not need to send ether or tokens to multiple addresses of course but the same technique may be adapted to generate a single transaction to deploy the contract for which the private key controlling the address is not known—in other words a keyless deployment using “Nick’s method”. A second advantage of this technique is that the address of a contract is deterministic. It is computed using the address from which the transaction originated and the nonce of the transaction. Specifically, the address is the `keccak256` hash of the owner’s address and the nonce, rlp-encoded with the first twelve bytes truncated. This means that the address of the contract is know in advanced and the address will be the same across all chains, thus solving the issue of looking up the address of the registry.

To build this transaction we all we need is to set the correct values in our message. Since this is a contract deployment the `to` address must be the zero address and the `value` should be zero as we do not want to send ether to the contract and the nonce should be zero as well since this is the first (and only) transaction from the given address. The `data` is the compiled bytecode needed to deploy the contract and all that remains is the gas and gas price. The gas consumption can easily be computed using the `eth_estimateGas` call since we know the code which will be executed as part of the transaction. The gas price is a bit more tricky. If set too

low the transaction may never be picked up by miners and sit in the mempool until it is evicted. Setting the gas price to high and the deployment will be very costly. At this point in time since the gas price is part of the signed message, adjusting the gas price will modify the message and result in a new hash, thus changing the origin address of the transaction and by extension the address of the contract. The EIP1014 propose the creation of a `CREATE2` opcode expressly to handle this case. The `CREATE2` opcode is able to only consider the origin address, the actual initialization code and some salt value [Buterin, 2018]. Sadly it is not yet available at this time.

Lastly, all we have left is to set the `r`, `s` and `v` values. The value for `v` is trivially set to 27. The value for `r` is set to `0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798` and most importantly, `s` must be set to the value `0x0aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`. This is a predictable value, to convince everyone that no one holds the private key for the address derived from the transaction.

We are now all set and with the above value we can generate the contract deployment transaction and derive its sender, then we must send enough ether to the address, and then broadcast the transaction.

TODO add actual values

6.7.1 Vanity Address

A vanity address is the equivalent of a vanity license plate for addresses on a blockchain. One of the first notion of vanity address comes from a utility names “Vanitygen” which is a command line tool to generate a Bitcoin address matching a specific pattern by using brute-force [Jtibble, 2012]. This of course does not allow to obtain the private key for a specific address but it allows to obtain an address starting or ending with a few specific characters.

Because there is only one specific address for the ERC820 registry and because this address is the same for all chains, we all agreed it would be a nice touch to have an address starting with `0x820` for the registry.

There is another issue with those vanity generators, of course “Vanitygen” is for Bitcoin but similar tools exist for Ethereum too. However those tools generate the private keys for regular accounts and the whole reason of using keyless deployment is to not have the keys to control the registry. Thus those programs do not fulfill our needs. Nonetheless, we can keep the idea behind vanity generators and use the process for keyless deployment to easily come up with a simple script which generate addresses for our contract.

Indeed, we know that if we change any of the field of the deployment transaction then we change the hash of the signed message and if we change the hash message we change the origin address returned by `ecrecover`. If we change the origin address, we also change the address of the deployed contract which is computed in a deterministic fashion from the origin address and the nonce. The only question which remains is which field can be safely changed in the transaction. The `to` must be the zero address, the `nonce` must be 0, setting a `value` other than 0 is literally the equivalent of burning ether and the gas price and gas limit are set specifically to make sure the transaction does not consume too little gas and that it will be expensive enough to be considered. The only remaining value which may be changed is the `data` which contains the initialization code for the registry contract.

This initialization code is automatically generated by a compile and should not be modified. Nevertheless, the initialization code contains a copy of the bytecode of the registry and while

we do not want to modify the actual code of the registry, there is one fact which can help us: bytecode compiled using `solc`, the Solidity compiler includes in the bytecode, the hash of the metadata for the compiled contract as return by the standard output of `solc` [see Foundation, 2018b, Encoding of the Metadata Hash in the Bytecode]. The reason for this choice is to be able to link the metadata to the specific instance of the contract.

Among other fields, this metadata can contain the original source code of the contract. And this is the key point, we can modify a random comment at the beginning of the source file. This will modify the content field of the metadata which will result in a different hash for the metadata which will result in a slightly different contract bytecode which will result in a slightly different deployment code and thus a slightly different data field of the transaction and finally in a different hash for the transaction or message. Hence we have managed to change the message hash thus changing the spender address and by extension the contract address.

```
#! /bin/bash

rm -rf tmp
mkdir tmp
for i in {0..15}; do
  mkdir -p "./tmp/${i}/contracts"
  cp contracts/ERC820Registry.sol "./tmp/${i}/contracts/"
done

for VALUE in ‘seq 0 16 32768‘; do
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: ${VALUE}/1" tmp/0/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+1))/1" tmp/1/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+2))/1" tmp/2/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+3))/1" tmp/3/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+4))/1" tmp/4/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+5))/1" tmp/5/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+6))/1" tmp/6/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+7))/1" tmp/7/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+8))/1" tmp/8/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+9))/1" tmp/9/contracts/
      ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+10))/1" tmp/10/contracts
      /ERC820Registry.sol
  sed -i ’’ -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+11))/1" tmp/11/contracts
      /ERC820Registry.sol
```

```
23  sed -i '' -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+12))/1" tmp/12/contracts
        /ERC820Registry.sol
24  sed -i '' -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+13))/1" tmp/13/contracts
        /ERC820Registry.sol
25  sed -i '' -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+14))/1" tmp/14/contracts
        /ERC820Registry.sol
26  sed -i '' -Ee "s/^\/\/ IV:.+$/\/\/ IV: $((${VALUE}+15))/1" tmp/15/contracts
        /ERC820Registry.sol
27
28  pushd ./tmp/0 > /dev/null
29  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
30    --insert-file-names none --output-artifacts-dir artifacts --quiet &
31  popd > /dev/null
32  pushd ./tmp/1 > /dev/null
33  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
34    --insert-file-names none --output-artifacts-dir artifacts --quiet &
35  popd > /dev/null
36  pushd ./tmp/2 > /dev/null
37  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
38    --insert-file-names none --output-artifacts-dir artifacts --quiet &
39  popd > /dev/null
40  pushd ./tmp/3 > /dev/null
41  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
42    --insert-file-names none --output-artifacts-dir artifacts --quiet &
43  popd > /dev/null
44  wait
45  pushd ./tmp/4 > /dev/null
46  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
47    --insert-file-names none --output-artifacts-dir artifacts --quiet &
48  popd > /dev/null
49  pushd ./tmp/5 > /dev/null
50  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
51    --insert-file-names none --output-artifacts-dir artifacts --quiet &
52  popd > /dev/null
53  pushd ./tmp/6 > /dev/null
54  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
55    --insert-file-names none --output-artifacts-dir artifacts --quiet &
56  popd > /dev/null
57  pushd ./tmp/7 > /dev/null
58  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
```

```
    commit.e67f0147" \
59    --insert-file-names none --output-artifacts-dir artifacts --quiet &
60  popd > /dev/null
61  wait
62  pushd ./tmp/8 > /dev/null
63  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
64    --insert-file-names none --output-artifacts-dir artifacts --quiet &
65  popd > /dev/null
66  pushd ./tmp/9 > /dev/null
67  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
68    --insert-file-names none --output-artifacts-dir artifacts --quiet &
69  popd > /dev/null
70  pushd ./tmp/10 > /dev/null
71  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
72    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
73  popd > /dev/null
74  pushd ./tmp/11 > /dev/null
75  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
76    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
77  popd > /dev/null
78  wait
79  pushd ./tmp/12 > /dev/null
80  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
81    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
82  popd > /dev/null
83  pushd ./tmp/13 > /dev/null
84  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
85    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
86  popd > /dev/null
87  pushd ./tmp/14 > /dev/null
88  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
89    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
90  popd > /dev/null
91  pushd ./tmp/15 > /dev/null
92  npx solcpiler -i ./contracts/ERC820Registry.sol --solc-version="v0.4.24+
        commit.e67f0147" \
93    --insert-file-names none --output-artifacts-dir artifacts  --quiet &
94  popd > /dev/null
95  wait
96
```

```
97   node mine-info.js "${VALUE}" | tee -a addrs.txt
98
99   rm -rf tmp/0/build tmp/0/artifacts \
100    tmp/1/build tmp/1/artifacts \
101    tmp/2/build tmp/2/artifacts \
102    tmp/3/build tmp/3/artifacts \
103    tmp/4/build tmp/4/artifacts \
104    tmp/5/build tmp/5/artifacts \
105    tmp/6/build tmp/6/artifacts \
106    tmp/7/build tmp/7/artifacts \
107    tmp/8/build tmp/8/artifacts \
108    tmp/9/build tmp/9/artifacts \
109    tmp/10/build tmp/10/artifacts \
110    tmp/11/build tmp/11/artifacts \
111    tmp/12/build tmp/12/artifacts \
112    tmp/13/build tmp/13/artifacts \
113    tmp/14/build tmp/14/artifacts \
114    tmp/15/build tmp/15/artifacts
115
116 done
```

Listing 6.2. The bash script used to generate a vanity address for the ERC820 regsitry.

The listing 6.2 shows the rather crude bash implementation which was used to modify and recompile the ERC820 registry contract with a different value in a comment and the print all the values for which the contract address begins with `0x820`. This implementation tries sixteen different possibilities per iteration and call a Javascript program on line 97—shown in in the listing 6.3 program—which is capable of finding the address of the contract and print it together with the value if the address begins with a `0x820` for the sixteen contracts at each iteration of the loop.

```
1  const EthereumTx = require('ethereumjs-tx');
2  const EthereumUtils = require('ethereumjs-util');
3
4  const offeset = parseInt(process.argv[2]);
5  const rawTx = {
6      nonce: 0,
7      gasPrice: 100000000000,
8      gasLimit: 800000,
9      value: 0,
10     data: undefined,
11     v: 27,
12     r: '0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798',
13     s: '0x0aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
14 };
15
16 for (let i = 0; i < 16; i++) {
17   const code = '0x' + require('./tmp/${i}/artifacts/ERC820Registry.json').
         compilerOutput.evm.bytecode.object;
```

```
18   rawTx.data = code
19   const tx = new EthereumTx(rawTx);
20   const contractAddr = EthereumUtils.toChecksumAddress(
21     '0x' + EthereumUtils.generateAddress('0x' + tx.getSenderAddress().
         toString('hex'), 0 ).toString('hex')
22   );
23   if (contractAddr.startsWith('0x820')) {
24     console.log(`${offeset + i} -> ${contractAddr}`);
25   }
26 }
```

Listing 6.3. The Javascript program called by the ERC820 Vanity generator to find the address of a version of the compiled 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, 49

API

Application Programming Interface. 49

DApp

Decentralized Application. 8, 9, 20, 22, 28, 29, 49

DEX

Decentralized Exchange. 18, 22, 49

DNS

Domain Name System, The Internet's system for converting alphabetic names into numeric IP addresses [PCMag.com]. 49

ECDSA

Elliptic Curve Digital Signing Algorithm. 33, 34, 49

EIP

Ethereum Improvement Proposal. 8, 23, 24, 35, 49

ENS

The Ethereum Name Service, the equivalent of DNS for Ethereum addresses allwoing to resolve names such as `revolution.eth` to `0x5ADF43DD006c6C36506e2b2DFA352E60002d22Dc`. 28, 49

ERC

Ethereum Request for Comments. 29, 49

EVM

Ethereum Virtual Machine. 5, 49

internet troll

A person who intentionally antagonizes others online by posting inflammatory, irrelevant, or offensive comments or other disruptive content [Merriam-Webster.com]. 24, 49

LIFO

Last-In-First-Out. 5, 49

multsig wallet

A multi-signature (multisig) wallet is a wallet requiring the signatures of more than one key to authorize a transaction. Usually it requires M out of N signatures, where $M \leq N$. 27, 34, 49

UX

User eXperience. 1, 14, 22, 49

Web3

Web3 often refers to `web3js`, the Javascript implementation of the ethereum JSON-RPC Application Programming Interface (API). It may also refer to other implementation in different languages. Overall it is the technology aiming to build the next and more decentralized version of the web 2.0 we know today. 49

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. 19–21, 23, 30, 31, 34, 35, 49

Bibliography

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

Jordi Baylina and Jorge Izquierdo. ERC672: Reverseens pseudo-introspection, or standard interface detection, July 2017. URL https://github.com/ethereum/EIPs/issues/672.

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

Vitalik Buterin. ERC724: Proposed erc20 change: make 18 decimal places compulsory, September 2017. URL https://github.com/ethereum/EIPs/issues/724.

Vitalik Buterin. EIP1014: Skinny CREATE2, May 2018. URL https://github.com/ethereum/EIPs/issues/1014.

Jacques Dafflon. Add tokensToSend callback, February 2018a. URL https://github.com/jacquesd/ERC777/issues/23.

Jacques Dafflon, February 2018b. URL https://github.com/jacquesd/ERC777/issues/23#issuecomment-363491500.

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

Jacques Dafflon, Jordi Baylina, and Thomas Shababi. ERC777: A new advanced token standard (original issue), November 2017b. URL https://github.com/ethereum/EIPs/issues/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.

Nick Johnson. How to send ether to 11,440 people, August 2016. URL https://medium.com/@weka/how-to-send-ether-to-11-440-people-187e332566b7.

Jtibble. How to send ether to 11,440 people, November 2012. URL https://en.bitcoin.it/wiki/Vanitygen.

Merriam-Webster.com. "troll". URL https://www.merriam-webster.com/dictionary/troll.

PCMag.com. "domain name service". URL https://www.pcmag.com/encyclopedia/term/41620/dns.

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.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version e94ebda - 2018-06-05, June 2018. URL https://github.com/ethereum/wiki/wiki/White-Paper.

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