Resolving the Multiple Withdrawal Attack on ERC20 Tokens

Reza Rahimian, Jeremy Clark, Shayan Eskandari
Concordia University

***Abstract*—Custom tokens are an integral component of decentralized applications (dapps) deployed on Ethereum and other blockchain platforms. For Ethereum, the ERC20 standard is a widely used token interface and is interoperable with many existing dapps, user interface platforms, and popular web applications (*e.g.,* exchange services). An ERC20 security issue, known as the *multiple withdrawal attack*, was raised on GitHub and has been open since October 2017. The issue concerns ERC20's defined method `approve()` which was envisioned as a way for token holders to give permission for other users and dapps to withdraw a capped number of tokens. The security issue arises when a token holder wants to adjust the amount of approved tokens from N to M. If malicious, a user or dapp who is approved for N tokens can front-run the adjustment transaction to first withdraw N tokens, then allow the approval to be confirmed, and withdraw an additional M tokens afterwards. In this paper, we evaluate 10 proposed mitigations for this issues and find that no solution is fully satisfactory. We then propose 2 new solutions that mitigate the attack, one of which fully fulfills constraints of the standard, and we also show a general limitation in addressing this issue from ERC20's approve method.**

***Index Terms*—Ethereum; ERC20 tokens; Blockchain;**

1. Introduction

Ethereum is a public blockchain proposed in 2013, deployed in 2015 [1], and has the second largest market cap at the time of writing[1]. It has a large development community which track enhancements and propose new ideas.[2] Ethereum enables decentralized applications (dapps) to be deployed and executed. dapps can accept and transfer Ethereum's built-in currency (ETH) or might issue their own custom currency-like tokens for specific purposes. Tokens might be currencies with different properties than ETH, they may be required for access to a dapp's functionality or they might represent ownership of some off-blockchain asset. It is beneficial to have interoperable tokens with other dapps and off-blockchain webapps, such as exchange services that allow tokens to be traded.

Toward this goal, the Ethereum project accepted a proposed standard (called ERC20 [2]) for a set of methods which standard-compliant tokens should implement. In terms of object oriented programming, ERC20 is an interface that defines abstract methods (name, parameters, return types) and provides guidelines on how the methods should be implemented, however it does not provide an actual concrete implementation (see Figure 1).

```
contract ERC20Interface {
    function totalSupply() public view returns (uint256);
    function balanceOf(address _tokenOwner)
        public view returns (uint256 tokens);
    function transfer(address _to, uint256 _tokens)
        public returns (bool success);
    function approve(address _spender, uint256 _tokens)
        public returns (bool success);
    function transferFrom(address _from, address _to, uint256 _tokens)
        public returns (bool success);
    function allowance(address _tokenOwner, address _spender)
        public view returns (uint256 remaining);

    event Transfer(address indexed _from,
                   address indexed _to,
                   uint256 _tokens);
    event Approval(address indexed _tokenOwner,
                   address indexed _spender,
                   uint256 _tokens);
}
```

Figure 1. The ERC20 standard defines 6 methods and 2 events that must be implemented. Using `approve` and `transferFrom` in the existence of a race condition may lead to a *"multiple withdrawal attack." Note that* `transferFrom` *augments the more basic* `transfer` *method.*

Since the introduction of ERC20 in November 2015, several vulnerabilities have been discovered. In October 2017, a security issue called *"Multiple Withdrawal Attack"* was opened on GitHub [3], [4]. The attack originates from two methods in the ERC20 standard for approving and transferring tokens. The use of these functions in an adverse environment (*e.g.,* front-running [5]) could result in more tokens being spent than what was intended. This issue is still open and several solutions have been made to mitigate it. The authors of the ERC20 standard [2] reference two sample implementations: OpenZeppelin [6] and ConsenSys [7]. OpenZeppelin mitigates the attack by introducing two additional methods to replace the `approve` method, and the ConsenSys implementation does not attempt to resolve the attack. Additional implementations have a variety of different trade-offs in mitigating the issue (see Section 3).

Contributions. In this paper, we evaluate 10 proposed mitigations for the multiple withdrawal attack. We develop a set of criteria that encompass backwards compatibility, interoperability, adherence to the ERC20 standard, and attack mitigation. The summary is provided in Figure 2. Since no mitigation is fully satisfactory, we develop two additional solutions based on the *Compare and Set (CAS)* pattern[8]—a widely used lock-free synchronization strategy that allows comparing and setting values atomically. We study in detail possible implementations of ERC20's `approve` and `transferFrom` methods. We argue that a CAS-based approach can never adequately deploy a secure `approve` method while adhering to the ERC20 standard. We then propose a secure implementation of the `transferFrom` method that mitigates the attack and fully satisfies the ERC20 standard.

1. [2019-02-11] https://coinmarketcap.com/currencies/ethereum/
2. [2019-02-11] https://www.coindesk.com/data

#	Proposed solution	Is it backward compatible?	Does it remediate vulnerable functions?	Does it allow non-zero allowances?	Does it allows zero token transfers?	Does it mitigate the attack?
Suggested Solutions						
1	Enforcement by UI	✓ It does not change any code	✗ It does not change any code	✓ By default approve method	✓ By default transferFrom method	✗ Race condition can occur
2	MiniMe Token	✓ It adds only one line to approve method	✗ Only forces allowance to be zero before non-zero values	✓ If it is already zero, otherwise in two calls	✓ By default transferFrom method	✗ Race condition can occur
3	Minimum viable token	✗ It does not implement vulnerable functions	✗ It does not implement approve function	✗ It does not implement approve function	✗ It does not implement transferFrom function	✓ By not addressingvulnerable functions
4	Approving trusted parties	✓ It does not change any code	It depends on code verification	✓ By default approve method	✓ By default transferFrom method	✓ It is not a universal solution
5	Monolith DAO	✗ It adds two new functions	✗ It does not change any code	✗ It adjusts allowance	✓ By default transferFrom method	✓ By using two new methods
6	Alternate approval function	✗ It adds one new function	✗ It does not change any code	✓ By using new method	✓ By default transferFrom method	✓ By using new method
7	Changing ERC20 API	✗ It adds new overloaded approve method	✓ By new method with three parameters	✓ By using new method	✓ By default transferFrom method	✓ By using new method
8	New token standards	✗ It introduces new API	✓	✓	✓	✓
9	Detecting token transfers	✓ It adds two lines to approve method	✓	✗ It locks down allowance in case of any token transfer	✓ By default transferFrom method	✓ It blocks legit and non-legit allowances as well
10	Keeping track of remaining tokens	✓ It adds three lines to the approve method	✓	✓	✓ By default transferFrom method	✗ Race condition can occur
New Proposals						
11	Proposal 1: Securing approve method	✓ It adds new codes to the approve method	✓	✗ It adjusts the allowance	✓	✓
12	Proposal 2: Securing transferFrom method	✓ It adds new codes to transferFrom method	✓ It secures transferFrom method	✓ By default approve method	✓	✓

Figure 2. Comparison of 10 proposals and 2 contributed by this paper. In our first proposal, CAS is used to mitigate the attack by comparing transferred tokens with new allowance. It is not fully ERC20-compliant since the allowance result does not always match what is requested. In our second proposal, a new local variable is defined to keep track of transferred tokens and prevents transfers in the case of already transferred tokens.

2. Preliminaries

2.1. How the *multiple withdrawal attack* works

According to the ERC20 API definition, the `approve` function[3] allows a spender (*e.g.,* user, wallet or other smart contracts) to withdraw up to an allowed amount of tokens from token pool of the approver. If this function is called again, it overwrites the current allowance with the new input value. On the other hand, the `transferFrom` function allows the spender to actually transfer tokens from the approver to anyone they choose (importantly: not necessarily themselves). The contract updates balance of transaction parties accordingly.

An adversary can exploit the gap between the confirmation of the `approve` and `transferFrom` functions since the `approve` method replaces the current spender allowance with the new amount, regardless of whether the spender already transferred any tokens or not. This functionality of the `approve` method is shaped by the language of the standard and cannot be changed. Furthermore, while variables change and events are logged, this information is ambiguous and cannot fully distinguish between possible traces. Consider the following illustration:

3. We use the term method and function interchangeably.

1) Alice allows Bob to transfer N tokens on her behalf by broadcasting `approve(_Bob, N)`.
2) Later, Alice decides to change Bob's approval from N to M by broadcasting `approve(_Bob, M)`.
3) Bob notices Alice's second transaction after it is broadcast to the Ethereum network but before it is added to a block.
4) Bob front-runs (using an asymmetric insertion attack [5]) the original transaction with a call to `transferFrom(_Alice, _Bob, N)`. If a miner is incentivized (*e.g.,* by Bob offering high gas) to add this transaction before Alice's, it will transfer N of Alice's tokens to Bob.
5) Alice's transaction will then be executed which changes Bob's approval to M.
6) Bob can call `transferFrom` method again and transfer M additional tokens by broadcasting `transferFrom(_Alice, _Bob, M)`.

In summary, in attempting to change Bob's allowance from N to M, Alice makes it possible for Bob to transfer N+M of her tokens. We operate on the assumption that a secure implementation would prevent Bob from withdrawing Alice's tokens multiple times when the allowance changes from N to M (see Figure 3).

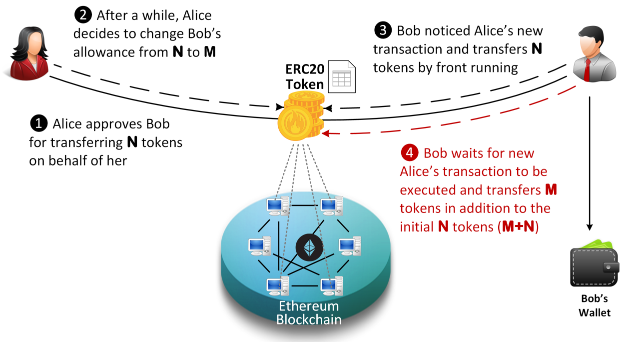


Figure 3. Possible multiple withdrawal attack in ERC20 tokens when Alice changes Bob's allowance from N to M. By front-running, Bob is able to move N+M tokens from token pool of Alice. Solutions consider Bob's initial transfer of N tokens (in step 3) to be legitimate and try to prevent the second transfer of the additional M tokens in step 4.

2.2. Why mitigation is important

ERC20 tokens are important component of Ethereum's supplementary financial system. ERC20 tokens have many financial (as well as non-financial) uses and could hold considerable value (potentially exceeding the value of Ether itself) and there has been more than 64,000 functional ERC20 tokens networks as of early 2019 [9].Furthermore, ERC20 tokens that have already been issued cannot easily migrate to a new secure implementation and should these tokens appreciate in value in the future. It also serves as basis for other extended standards, such as ERC-223, ERC-621, and even ERC-777 is backward compatible with ERC20 interface [10]. Finally, firms that hold ERC20 tokens require assurance of their security, particularly in the case that they require their financial statements to be audited—an issue like this could lead to further hesitation by auditors.

2.3. Where to prevent this attack

There are a few logical places to address this attack. Ideally the token author (instead of the token holders) would mitigate the attack within the ERC20 implementation itself. Since two methods are involved in the attack, it could be addressed within the `approve` and/or `transferFrom` method. By contrast, token owners have no control over the implementation of the contract, and so token owners are relegated to mitigating the attack by carefully monitoring the contract around the time allowance adjustments are made.

Prevention by token owners. Consider Alice, a token holder using a web app (*e.g.,* a user interface deployed with web3) to adjust Bob's allowance. If this UI is written to the ERC20 standard, Alice will only have `approve` available to her. The ERC20 authors [2] advise Alice against directly changing Bob's allowance from N to M. Instead, she should set the approval to 0 and then it to M. Presumedly, Alice will not do the second approval (setting it from 0 to M) if she sees that Bob withdraws N tokens before her N to 0 adjustment is confirmed.

How will Alice know whether Bob withdraws first? The answer depends on how deeply her webapp can monitor the blockchain. One option is to monitor the variable that records Bob's allowance (technically an on-blockchain helper dapp could also do this). However if she sees Bob's allowance at 0 after initiating the first adjustment, it does not tell her that Bob did not withhdraw N—both methods result in Bob having an allowance of 0 so either (or both) could have been executed.

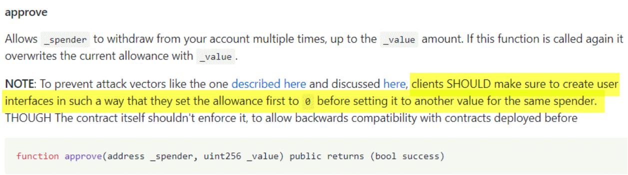
Next, she might rely on events passed from the contract to her web3 app. `Transfer` events as specified in ERC20 will log the transfer parameters (*i.e.,* `address _from, address _to, uint256 _tokens`). Alice's webapp could filter the events and only display transfers matching her address in the `_from` field. The displayed transfers will include any transfer Bob makes, however Bob can provide any address in `_to`, not just his own, and the event does not report who authorized the transaction (*i.e.,* `msg.sender`). If Alice has many authorizations, she cannot determine if a transfer was initiated by Bob or someone else she has authorized.[4] Therefore Alice cannot always unambiguously rely on events to determine Bob did not transfer funds. If Alice's web app is beyond web3 and runs a full node maintaining blockchain state, it could correctly detect and attribute all transfers initiated by Bob.

The takeaway from all these options is that prevention by token owners has some undesirable properties: (1) it splits adjustments into two transactions, (2) because the first transaction needs to be confirmed before the second is initiated, it takes time to complete, and (3) to precisely mitigate this attack, a heavyweight web app is needed to inspect deeper than variable state changes and events. For these reasons, we concentrate on mitigating the attack in the contract itself. If mitigation at the contract level works, allowances can be adjusted with a single function call from any existing lightweight ERC20 user interface, and no additional monitoring of the contract is necessary.

Prevention by token author in `approve`. The next logical place to tackle multiple withdrawal is in the implementation of the `approve` method. In particular, an `approve` implementation might be engineered to fail under the conditions of multiple withdrawal, to adjust the approval amount, treat adjustments as relative offsets from the current amount, or other techniques. As we review the 10 solutions, we will see different proposals along these lines, as well as our own proposal in Section 4.1. For now, we emphasize that adherence to the standard is a core challenge as it unequivocally states that: "If this function is called again, it overwrites the current allowance with `_value`" [2].

Prevention by token author in `transferFrom`. Recall from Figure 3 that step 4 is the offending function call and it is to `transferFrom`. If we add new state to the contract to track the number of tokens that have been transferred, we can allow approval to work exactly as specified while interpreting it as a "lifetime" allowance. We will explain this in more detail in Section 4.2.

4. Even if Bob is listed in `_to`, another authorized spender might have transfered tokens to Bob to make Alice believe Bob attempted a multiple withdrawal when he actually did not.


approve

Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.

NOTE: To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to `0` before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

Figure 4. Recommendation of ERC20 standard to mitigate multiple withdrawal attack by enforcement in UI.

2.4. What an ideal solution looks like

We prioritize adherence to the ERC20 standard. While deviating from the standard might become acceptable if there is no possible way to conform with it and maintain security, we consider that a last resort. Indeed, as we will show, it is possible to secure an ERC20 contract within the constraints of the standard, which we summarize here [2]:

1) The input to `approve` is a new allowance and not a relative adjustment.
2) The result of `approve` will overwrite the current allowance with the new allowance.
3) A call to `transferFrom` on an input of 0 tokens will execute as a normal transfer and emit a `Transfer` event.
4) A spender can call `transferFrom` multiple times up to the allowed amount.
5) Transferring up to any initial allowance is always a legitimate transfer.
6) An ideal solution cannot rely on overloading existing methods or introducing new methods outside of ERC20, as existing ERC20 dapps and web apps would have to be modified to interoperate.
7) A solution must eliminate all race conditions.

3. Evaluating Proposed Solutions

In this section, we evaluate 10 solutions that have been proposed by Ethereum community—mostly from developers on GitHub—to address the multiple withdrawal attack. We examine each solution in detail and evaluate them against the criteria established in the previous section (see Section 2.4). The summary is presented in Table 2.

3.1. Enforcement by User Interface (UI)

The first solution is enforcement at the user interface level. We discussed this previously in Section but we reiterate the main points here again. The exact recommendation from the ERC20 standard is shown in Figure 4 and is essentially to set an allowance to zero before any non-zero values. Presumedly, it will also enforce that the new approval is not allowed if proceeded by a token transfer by the approved spender. We consider the UI to be a lightweight web app that can reference a contract's state variables and emitted events but does not maintain a full copy of the blockchain. Consider (again) the most basic attack sequence:

1) Alice allows Bob to transfer N of Alice's tokens.
2) Alice's client broadcasts an allowance of 0 for Bob.
3) Bob broadcasts a competing transaction to transfer N of Alice's tokens.
4) Bob's transaction front-runs Alice's, is confirmed, and sets Bob's allowance to 0 (from N).
5) Alice's transaction is confirmed and sets Bob's allowance to 0 (from 0).
6) Alice's client broadcasts an allowance of M for Bob.
7) Alice's second transaction is confirmed and sets Bob's allowance to M.
8) Bob transfers M of Alice's tokens for a total of N+M tokens.

The key mitigation to this attack is for Alice's client to pause at step 6 and determine if a transaction sequence like 3&4 has occurred or not. This cannot be determined by monitoring the integer that records Bob's allowance because it will be 0 regardless of whether 3&4 occurred or not. It also cannot always be determined by monitoring the events omitted from the contract. Step 4 will log a transfer from Alice's address to Bob's address of N tokens. If no event is omitted, Alice can know for certain no transfer was made. However if an event is emitted, Alice must decide it was a transfer initiated by Bob or a transfer by someone else she has authorized. If she has not authorized anyone else, she can know for certain it was Bob. However if she has a busy account with multiple authorized spenders of her tokens, the event is not verbose enough to determine what happened. Importantly, it does not record who initiated the transfer, only who received the tokens, and these are not necessarily the same entity. Bob can send Alice's tokens to his accomplice Carol, or some other authorized spender can send Alice's tokens to Bob (which looks like Bob is attacking when he is not). Since a UI is automated and does not use human discretion, it cannot decide circumstantially whether something looks like an attack or not — it must either specify exact rules which it cannot do precisely for ambiguous events or it can ask for Alice's human input which introduces usability issues. In conclusion, it is better for enforcement to happen at the contract level. The trick for approving 0 and then approving N can be implemented.

Interestingly, OpenZeppelin example implements a workaround in contract level by adding extra functionality that are not part of the standard and make it inconsistent with the standard UI recommendations. Their solution is to add two new functions for increasing and decreasing the approved value, hence not fixing the standard functions [6].

3.2. Approving trusted parties

Approving token transfer to parties that we trust, reduces risk of impacting by the attack. Non-upgradable smart contracts or trusted accounts can be considered safe for delegating token transfers on behalf of us. Because they do not contain any logic to take advantage of this vulnerability. However, upgradable smart contracts may add new code to their new versions that needs code reverification before approving token transfers. Similarly, approving token transfer to people that we trust could be considered as a mitigation plan. Nonetheless, this solution would have limited use cases and it could not be considered as a universal preventive approach.

```
221        // To change the approve amount you first have to reduce the addresses`
222        //  allowance to zero by calling `approve(_spender,0)` if it is not
223        //  already 0 to mitigate the race condition described here:
224        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
225        require((_amount == 0) || (allowed[msg.sender][_spender] == 0));
```

Figure 5. MiniMeToken added code to approve method for allowing non-zero allowance values if it is already set to zero.

State	Input value (_value)	Current spender's allowance	Approve function result	New spender's allowance
1	Zero	Non-zero	Set to _value	0
2	Zero	Zero	Set to _value	0
3	Non-zero	Zero	Set to _value	_value
4	Non-zero	Non-zero	No result	No change

Figure 6. Functionality of default approve method in MonolithDAO token that enforces setting spender's allowance to zero before any non-zero values. It implements ERC20 recommendation for changes allowance from N to M in two steps (N→0→M).

3.3. MiniMeToken

MiniMeToken[11] follows ERC20 recommendation for setting allowance to zero before non-zero values. Token holder needs to execute two transactions, setting allowance from N to zero and then from 0 to M. MiniMeToken added a new line of code to the approve method for preventing the attack by enforcing this rule (see Figure 5). The red clause in line 225 (_amount == 0) allows setting of approval to 0 and blue condition checks allowance of _spender to be 0 before setting to non-zero values (*i.e.,* If _spender allowance is 0 then allows non-zero values). Similar to 3.1, this solution will not prevent Bob from transferring N+M tokens. Because Alice would not be able to distinguish whether N tokens have been already drained by Bob or not. Considering the below scenario:

1) Alice decides to set Bob's allowance to 0.
2) Bob front-runs Alice's transaction and his allowance sets to 0 after transferring N tokens.
3) Alice's transaction is executed and sets Bob's allowance to 0 (Red clause passes sanity check).
4) Alice checks Bob's allowance and she will find it 0, so, she can not determine whether this was because of her transaction or Bob already transferred N tokens.
5) Alice considers that Bob has not transferred any tokens and allows him for transferring new M tokens.
6) Bob is able to transfer new M tokens and N+M in total.

3.4. MonolithDAO

MonolithDAO Token[12] implements two additional functions for allowance increase[5] or decrease[6]. The default approve function has additional codes to enforce the owner for setting allowance to zero before non-zero values. It allows non-zero spender's allowance if it is already set to zero (see Figure 6). If the current spender's allowance is non-zero, decreaseApproval and increaseApproval functions have to be used instead of standard approve method as explained below:

1) Alice allows Bob to transfer N tokens by calling approve(_Bob, N). Alice used the default approve function consciously since current Bob's allowance is 0. So, he checked Bob's allowance before calling approve method.
2) After a while, Alice decides to decrease Bob's approval by M and runs decreaseApproval(_Bob, M).
3) Bob notices Alice's second transaction and front runs it by executing transferFrom(_Alice, _Bob, N).
4) Bob's transaction will be executed first and transfers N token to his account and the his allowance becomes 0 as result of this transfer.
5) Alice's transaction is mined after Bob's transaction and tries to decrease Bob's allowance by M. If Bob had already transferred more than M tokens, new Bob's allowance becomes negative and it fails the transaction. So, the transaction does not change Bob's remaining allowance and he would be able to transfer the rest (which is legitimate transfer since Alice has already approved it). If Bob had transferred less than M tokens, the new allowance will be applied and reduces Bob's allowance by M.

Although these two new complementary functions prevent the attack, they have not been defined in the initial specifications of ERC20 standard. Therefore, they can not be used by smart contracts that are already deployed on the Ethereum network and still call approve method for setting new allowance—and not increaseApproval or decreaseApproval. Moreover, ERC20 specifications does not define any increase or decrease of allowance. It only allows setting new allowances without adjustment. For example, if Alice has approved Bob for 100 tokens and wants to set it to 80, the new allowance should be 80 while using decrease methods will set it to 20 (100 - 80 = 20). Comparatively, increase method sets new allowance to 180 while it has to set it to 80 again to be in-compliant with ERC20 specification. For these reasons, this solution would not be compatible with ERC20 standard and only is usable if approver or smart contract are aware of these supplementary methods.

3.5. Alternate approval function

Another suggestion[13] is to move security checks to another function called safeApprove[7] that compare current and new allowance value and sets it if has not been already changed. By using this function, Alice uses the standard approve function to set Bob's allowance to 0 and for new approvals, she has to use safeApprove function. safeApprove takes the current expected approval amount as input parameter and calls approve

5. increaseApproval(address _spender, uint _addedValue)
6. decreaseApproval(address _spender, uint _subtractedValue)
7. safeApprove(address _spender, uint256 _currentValue, uint256 _value

method if previous allowance is equal to the current expected approval. By using this function, Alice will have one step more to read the current allowance and pass it to the new `safeApprove` method. Although this approach mitigates the attack by using CAS pattern[8], however, it is not backward compatible with already implemented smart contracts due to their unawareness of this new complementary function. In other words, the new `safeApprove` method is not defined in ERC20 standard and existing smart contracts would not be able to use this new safety feature.

3.6. Detecting token transfers

In order to set new allowance atomically, tracking of transferred tokens is required to detect token transfers before setting new allowances. If `approve` method reveals any transferred tokens due to front running, it throws an exception without setting new allowance. As suggested by [14], a flag can be used to detect whether any token has been transferred or not. `transferFrom` method sets this flag to `true` in case of any token transfer. `approve` method checks the flag to be `false` before allowing new approvals. This approach requires new data structure to keep track of used/transferred tokens for each spender. It can prevent front-running in the below scenario:

1) Alice runs `approve(_Bob, N)` to allow Bob for transferring N tokens. Since Bob's initial allowance is 0 and his corresponding flag=`false`, then sanity check passes and Bob's allowance sets to N (line 16).
2) Alice decides to set Bob's allowance to 0 by executing `approve(_Bob, 0)`.
3) Bob front-runs Alice's transaction and transfers N tokens. Then, `transferFrom` turns his flag to `true`.
4) Alice's transaction is mined and passes sanity check because passed value is 0 in line 15.
5) Bob's allowance is set to 0 while his flag remains `true`. (`approve` method does not flip spender flags.)
6) Alice wants to change Bob's allowance to M by executing `approve(_Bob, M)`. Since Bob already transferred N tokens (his flag=`true`), then transaction fails.
7) Bob's allowance does not change and he cannot move more tokens than initially allowed.

```
14      function approve(address _spender, uint _value) public returns (bool success) {
15          require((_value == 0) || (allowed[msg.sender][_spender].amount == 0 &&
                                     !allowed[msg.sender][_spender].used));
16          allowed[msg.sender][_spender].amount=_value;
17          Approval(msg.sender,_spender,_value);
18          return true;
19      }
```

Figure 7. `approve` method needs to be modified by adding a line of code like `allowed[msg.sender][_spender].used = false;` between lines 16 and 17 to unlock spender flag for the next legitimate change. However, this change makes attack mitigation ineffective.

Although this approach mitigates the attack, but it prevents any further legitimate approvals as well. Considering a scenario that Alice rightfully wants to increase Bob's allowance from N to M (two non-zero values). If Bob had already transferred number of tokens, Alice would not be able to change his approval. Because Bob's flag is set to `true` and line 15 does not allow changing allowance by throwing an exception (see Figure 7). Even setting allowance to 0 and then to M, does not flip the flag to `false` (There is no code for it in the `approve` method). So, It keeps Bob's allowance locked down and blocked further legitimate allowances. In fact, `approve` method needs a new code between lines 16 and 17 to set the flag to `false`. But it will cause another problem. After setting allowance to 0, spender flag becomes `false` and allows non-zero values even if tokens have been already transferred. Considering front-running by Bob in the below scenario:

1) Alice changes Bob's allowance from N to 0.
2) Bob transfers N tokens before allowance change and his `used` flag turns to `true`.
3) Alice's transaction is successful since `_value=0`. (The second condition is not evaluated although `used` flag is `true`).
4) Alice transaction turns `used` flag to `false` and sets Bob's allowance to 0.
5) Now Alice wants to set Bob's allowance from 0 to M, His flag is `false` and allowance is 0. So, Alice cannot distinguish whether Bob moved any token or not. Setting new allowance will allow Bob to transfer more tokens than Alice wanted.

In fact, resetting the flag in `approve` method will not fix the issue and makes attack mitigation ineffective. In short, this approach can not satisfy both legitimate and non-legitimate scenarios. Nevertheless, it is a step forward by introducing the need for a new variable to track transferred tokens.

3.7. Keeping track of remaining tokens

This approach[15] is inspired by the previous solution and keeping track of remaining tokens instead of detecting transferred tokens. `approve` method uses these variables to set allowance accordingly. It uses modified version of data structure that used in the previous solution for storing residual tokens (see Figure 8). At the first look, it seems to be a promising solution by setting approval to zero before non-zero values. However, the highlighted code in `approve` method resembles the situation that is explained in 3.1. In case of front-running, both `initial` and `residual` variables will be zero and it would not be possible for Alice to distinguish if any token transfer have occurred due to her allowance change or Bob token transfer. To make it more clear, considering the below scenario:

1) Bob's allowance is initially zero (`allowances[_Alice][_Bob].initial=0`) and his residual is zero as well (`allowances[_Alice][_Bob].residual=0`).
2) Alice allows Bob to transfer N tokens that makes `allowances[_Alice][_Bob].initial=N` and `allowances[_Alice][_Bob].residual=N`.
3) Alice decides to change Bob's allowance to M and has to set it to zero before any non-zero.

```
struct Allowance {
    uint initial;
    uint residual;
}

mapping(address => mapping(address => Allowance)) public allowances;

function approve(address spender, uint amount) public returns (bool) {
    Allowance storage _allowance = allowances[msg.sender][spender];

    // This test should not be necessary.
    uint spent = _allowance.initial > _allowance.residual
               ? _allowance.initial - _allowance.residual
               : 0;

    _allowance.initial = amount;
    _allowance.residual = spent < amount ? amount - spent : 0;

    Approval(msg.sender, spender, _allowance.residual);

    return true;
}
```

Figure 8. Keeping track of remaining tokens by introducing new data structure.

4) Bob noticed Alice's transaction for setting his allowance to zero and transfers N tokens in advance. Consequently, `transferFrom` function sets his residual to zero (`allowances[_Alice][_Bob].residual=0`)
5) Alice's transaction for setting Bob's allowance to zero is mined and sets `allowances[_Alice][_Bob].initial=0` and `allowances[_Alice][_Bob].residual=0` This is similar to step 1 that no token has been transferred. So, Alice would not be able to distinguish whether any token have been transferred or not.
6) Considering no token transfer by Bob, Alice approves Bob for spending new M tokens.
7) Bob is able to transfer new M tokes in addition to initial N tokens.

Someone may think of using `Transfer` event to detect transferred tokens or checking approver balance to see any transferred tokens. As explained in 3.1, using `Transfer` event is not sufficient in case of transferring tokens to a third party. Checking approver balance also would not be an accurate way if the contract is busy and there are lot of transfers. So, it would be difficult for the approver to detect legitimate from non-legitimate tokens transfers. Overall, this approach cannot prevent the attack.

3.8. Changing ERC20 API

As advised by [16], changing ERC20 API could secure `approve` method by comparing current allowance of the spender and sets it to new value if he has not already been transferred any tokens. This allows atomic compare and set of spender allowance to make the attack impossible. This approach needs a new overloaded `approve` method with three parameters in addition to the standard `approve` method with two parameters (see Figure 9).

In order to use this new method, smart contracts have to update their codes to provide three parameters instead of current two, otherwise any `approve` call will use the standard vulnerable version with two parameters. Moreover, one more call is required to read current allowance

```
// Standard ERC20 Approve Method
function approve(address _spender, uint256 _value) public returns (bool success)
{
    allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}

// Atomic "Compare And Set" Approve Method
function approve(address _spender, uint256 _currentValue, uint256 _newValue)
                                                public returns (bool success)
{
    if (allowed[msg.sender][_spender] != _currentValue) { return false; }

    allowed[msg.sender][_spender] = _newValue;
    emit Approval(msg.sender, _spender, _newValue);
    emit Approval(msg.sender, _spender, _newValue, _currentValue);
    return true;
}
```

Figure 9. Suggested ERC20 API Change by adding new `approve` method with three parameters to compare and set new allowance atomically.

and pass it to the new `approve` method. Additionally, new event definition needs to be added to ERC20 API to log an approval events with four arguments. For backward compatibility reasons, both three-arguments and new four-arguments events have to be logged. All of these changes makes this token contract incompatible with already deployed smart contracts. Hence, we would not consider it as a sustainable solution.

3.9. Minimum viable token

As suggested by the Ethereum Foundation[17], we can reduce the ERC20 standard to a set of core functionalities and implement only the essential methods. The attack can be side-stepped if methods like `approve` and `transferFrom` are simply not implemented (recall that `transferFrom` is in addition to the more commonly used `transfer`) or are implemented to always throw an exception and revert. Golem Network Token (GNT[8]) is one of these examples since it does not implement the `approve`, `allowance` and `transferFrom` functions. According to the ERC20 specification [2], these methods are not OPTIONAL and must be implemented. Moreover, ignoring them can cause failed function calls from smart contracts or web apps that expect these methods to work as specified. Therefore, we categorize this approach as successfully mitigating the attack but not offering interoperability.

3.10. New token standards

Minimum viable tokens could alternatively be considered a new non-ERC20 token (*cf.* ERC223). In fact, there are many ERC20 alternatives that extend or modify ERC20 for a variety of purposes, mostly around functionality but some address multiple withdrawals. We summarize the main proposals in Table 1.

Despite the enhancements of these new token standards for future deployments, ERC20 is ingrained in the community and industry with168,092 deployed tokens[9], many interoperable developer tools and libraries, and web platforms built on trading these tokens. Ideally, and the goal of this paper, a backward compatible solution could

8. https://etherscan.io/address/0xa74476443119A942dE498590Fe1f2454d7D4aC0d#code
9. https://etherscan.io/tokens, Accessed 18-Feb-2019

Token Standard	A description of non-compliance with ERC20
ERC 223 [18]	It does not implement ERC20 `approve` + `transferFrom` mechanism by assuming it potentially insecure and inefficient.
ERC 667 [19]	It solves the problem of transfer function in ERC223 (i.e., The need to implement `onTokenTransfer` routing in the receiving contract). So, it does address the attack and uses the same code as ERC223 with a supplementary function.
ERC 721 [20]	Unlike ERC20 tokens that share the same characteristics, ERC721 tokens represent a non-fungible tokens (NFT) which are unique and non-interchangeable with each other. In addition to this differences, ERC721 does not implement `transfer` method of ERC20 standard and introduces a safe transfer function defined as `safeTransferFrom`.
ERC 777 [21]	This standard does not use `transfer` and `transferFrom` methods of ERC20 and implements `send` and `operatorSend` instead. Moreover, costly `approve` + `transferFrom` mechanism is replaced by `tokensReceived` function. Therefore, it would not be backward compatible with ERC20 requirements. Nevertheless a token contract may implement both ERC20 and ERC777 in parallel which still requires attack mitigation.
ERC 827 [22]	It uses OpenZeppelin[6] ERC20 implementation and defines three new functions to allow users for transferring data in addition to value in ERC20 transactions. This feature enables ERC20 tokens to have the same functionality as Ether (transferring data and value). In fact, it extends functionality of ERC20 tokens and not addressing the attack.
ERC 1155 [23]	It is improved version of ERC721 by allowing each Token ID to represent a new configurable token type, which may have its own metadata, supply and other attributes. ERC1155 aimed to remove the need to "approve" individual token contracts separately. Therefore, it does not implement any code to address the vulnerability
ERC 1377 [24]	It implements `approve` method with three parameters in addition to the ERC20 default `approve` with two inputs. Additionally, it uses OpenZeppelin [6] approach for increasing and decreasing approvals. We would consider it as mix of MiniToken and OpenZeppelin approaches that we discussed before.

TABLE 1. EVALUATION OF STANDARD'S ADHERENCE TO ERC20 AND MITIGATION OF THE MULTIPLE WITHDRAWAL ATTACK.

be found that does not change the ERC20 API or require token migration to a new standard (which is not necessarily possible to do at the contract level). Like minimum viable tokens, we categorize these approaches potentially mitigating the attack (depending on which standard — see Table 1) but not offering interoperability.

4. New mitigations

By this point, we have discussed 10 solutions to the multiple withdrawal attack and we evaluated them in terms of compatibility with the standard and attack mitigation (recall the summary in Table 2). Since none of them precisely satisfy the constraints of ERC20 standard, we now propose two new solutions to mitigate the attack.

4.1. Proposal 1: Securing `approve` method

By implementing CAS [8] in `approve` method, new allowance can be set atomically after comparing with transferred tokens. This tracking requires adding a new variable to `transferFrom` method (see Figure 10). Since this is an internal variable, it is not visible to already deployed smart contracts and keeps the `transferFrom` function compatible. Similarly, a block of code is added to the `approve` function (see Figure 11) to work in both cases with zero and non-zero allowances. Added code to the `approve` function, compares new allowance—passed as `_tokens` argument to the function—with the current allowance of the spender and already transferred tokens—highlighted as `allowed[msg.sender][_spender]` and `transferred[msg.sender][_spender]`. Then it decides to increase or decrease current allowance based on this comparison. If the new allowance is less than initial allowance—sum of `allowance` and `transferred` variables—it denotes decreasing of allowance, otherwise increasing of allowance was intended. Modified `approve` function prevents the attack in either increasing or decreasing of the allowance.

Unlike other solutions, there is no need to set allowance from N to 0 and then to M. Token holder can

```
function transferFrom(address _from, address _to, uint256 _tokens)
public returns (bool success) {
    require(_to != address(0));
    require(balances[_from] >= _tokens);              // Checks enough tokens
    require(allowed[_from][msg.sender] >= _tokens); // Checks enough allowance
    balances[_from] = balances[_from].sub(_tokens);
    allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_tokens);
    transferred[_from][msg.sender] = transferred[_from][msg.sender].add(_tokens);
    balances[_to] = balances[_to].add(_tokens);
    emit Transfer(_from, _to, _tokens);
    return true;
}
```

Figure 10. Modified version of `transferFrom` for keeping track of transferred tokens per spender.

```
function approve(address _spender, uint256 _tokens) public returns (bool success) {
    require(_spender != address(0));
    uint256 allowedTokens = 0;
    uint256 initiallyAllowed = allowed[msg.sender][_spender].add(transferred[msg.sender][_spender]);

    //Aprover reduces allowance
    if (_tokens <= initiallyAllowed){
        if (transferred[msg.sender][_spender] < _tokens){ // If less tokens had been transferred.
            allowedTokens = _tokens.sub(transferred[msg.sender][_spender]); // Allows the rest
        }
    }
    //Approver increases allowance
    else{
        allowedTokens = _tokens.sub(initiallyAllowed);
        allowedTokens = allowed[msg.sender][_spender].add(allowedTokens);
    }

    allowed[msg.sender][_spender] = allowedTokens;
    emit Approval(msg.sender, _spender, allowedTokens);
    return true;
}
```

Figure 11. Added code block to `approve` function to prevent the attack by comparing and setting new allowance atomically.

directly change the allowance from N to M which is saving one transaction accordingly. The following examples make functionality of added codes more clear when decreasing or increasing allowance:

Scenario A. Alice approves Bob for spending 100 tokens and then decides to decrease it to 10 tokens.

1) Alice approves Bob for transferring 100 tokens.
2) After a while, Alice decides to reduce Bob's allowance from 100 to 10 tokens.
3) Bob noticed Alice's new transaction and transfers 100 tokens by front-running.
4) Bob's allowance is 0 and `transferred` is 100 (set by `transferFrom` function).
5) Alice's transaction is mined and checks initial allowance (100) with new allowance (10).

	Consumed Gas by the token		
Operation	TKNv1	TKNv2	Difference
Creating smart contract	1095561	1363450	25 %
Calling Approve function	45289	46840	4 %
Calling transferFrom function	44019	64705	47 %

Figure 12. Comparison of gas consumption between standard implementation of ERC20 token (TKNv1) and secured implementation of it (TKNv2).

6) As it is reducing, `transferred` tokens (100) is compared with new allowance (10). Since Bob already transferred more tokens, his allowance will be set to 0.
7) Bob is not able to move more than initial 100 approved tokens.

Scenario B. Alice approves Bob for spending 100 tokens and then decides to increase it to 120 tokens.

1) Alice approves Bob for transferring 100 tokens.
2) After a while, Alice decides to increase Bob's allowance from 100 to 120 tokens.
3) Bob noticed Alice's new transaction and transfers 100 tokens by front-running.
4) Bob's allowance is 0 and `transferred` is 100.
5) Alice's transaction is mined and checks initial allowance (100) with new allowance (120).
6) As it is increasing, new allowance (120) will be subtracted from transferred tokens (100).
7) 20 tokens will be added to Bob's allowance.
8) Bob would be able to transfer more 20 tokens (120 in total as Alice wanted).

In order to evaluate functionality of the new `approve` and `transferFrom` functions, we have implemented a standard ERC20 token (TKNv1[10]) along side the proposed ERC20 token (TKNv2[11]) on Rinkeby test network. Result of tests for different input values shows that TKNv2 can address multiple withdrawal attack by making front-running gain ineffective. Moreover, we compared these two tokens in term of gas consumption. TokenV2.`approve` function uses almost the same amount of gas as TokenV1.`approve`, however, gas consumption of TokenV2.`transferFrom` is around 47% more than TokenV1.`transferFrom` (see Figure 12). This difference is because of maintaining a new mapping variable for tracking transferred tokens. In term of compatibility, working with standard wallets (*e.g.,* MetaMask) have not raised any transfer issue. This shows compatibility of the token with existing wallets.

In summary, we could use CAS pattern to implement a secure `approve` method that can mitigate the attack effectively. However, it violates one of ERC20 specifications that says "If this function is called again it overwrites the current allowance with `_value`". This implementation of `approve` method adjusts allowance based on transferred tokens. Essentially, it would not be possible to secure the `approve` method without adjusting the allowance. Considering the below scenario:

1) Alice decides to change Bob's allowance from N to M (M less than N in this example).
2) Bob transfers N tokens by front running and `transferred` variable sets to N.
3) Alice's transaction is mined and `approve` method detects token transfer. If `approve` method does not adjust the allowance based on transferred tokens, it has to set it to M—to conform with the standard—which is allowing Bob to transfer more M tokens.

Therefore, `approve` method has to adjust the allowance according to transferred tokens, not based on passed input values to the `approve` method. Overall, there is no solution to secure `approve` method while adhering specification of ERC20 standard.

4.2. Proposal 2: Securing `transferFrom` method

As an alternative solution to Proposal 1, we can think of securing `transferFrom` method instead of `approve` function. As specified by ERC20 standard (see figure 13), the goal here is to prevent spender from transferring more tokens than allowed. Based on this assumption, we should not consider allowance as the main prevention factor. Transferred tokens can be considered as the main variable in our calculations. For example in the below situation, we can prevent the attack by securing `transferFrom` method and keeping `approve` function untouched to set allowance as specified by the token holder:

1) Alice allowed Bob for transferring 100 tokens and decides to set it to 70 after a while.
2) Bob front runs Alice's transaction and transfers 100 tokes (legitimate transfer).
3) Alice transaction is mined and sets Bob allowance to 70 by the default `approve` method.
4) Bob noticed new allowance and tries to move new tokens by running `transferFrom(_Bob,70)`. Since he already transferred more than 70, his transaction fails and prevents multiple withdrawal. Additionally, Bob's allowance stays as 70, although transferred tokens shows 100.

Here, `allowance` value can be considered as maximum allowance. It indicates that Bob is eligible to transfer up to specified limit if he has not already transferred any tokens. This impression is completely in accordance with ERC20 standard (see figure 13). In fact, there is no relation between allowance (`allowed[_from][msg.sender]`) and transferred tokens (`transferred[_from][msg.sender]`). The first variable shows maximum transferable tokens by a spender and can be changed irrelative to transferred tokens (*i.e.,* `approve` method does not check transferred tokens). If Bob has not already transferred that much of tokens, he would be able to transfer difference of it—`allowed[_from][msg.sender].sub(transfer`

10. https://rinkeby.etherscan.io/address/0x8825bac68a3f6939c296a40fc8078d18c2f66ac7
11. https://rinkeby.etherscan.io/address/0xf2b34125223ee54dff48f71567d4b2a4a0c9858b

transferFrom

Transfers `_value` amount of tokens from address `_from` to address `_to`, and MUST fire the `Transfer` event.

The `transferFrom` method is used for a withdraw workflow, allowing contracts to transfer tokens on your behalf. This can be used for example to allow a contract to transfer tokens on your behalf and/or to charge fees in sub-currencies. The function SHOULD `throw` unless the `_from` account has deliberately authorized the sender of the message via some mechanism.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
```

Figure 13. ERC20 `transferFrom` method definition that emphasizes on throwing an exception when the spender is not authorized to move tokens.

```
function transferFrom(address _from, address _to, uint256 _tokens) public returns (bool success) {
    require(_to != address(0));
    require(balances[_from] >= _tokens);                    // Checks if approver has enough tokens
    require(_tokens <= (
                    (allowed[_from][msg.sender] > transferred[_from][msg.sender]) ?
                     allowed[_from][msg.sender].sub(transferred[_from][msg.sender]) : 0)
                    );                                      // Prevent token transfer more than allowance

    balances[_from] = balances[_from].sub(_tokens);
    transferred[_from][msg.sender] = transferred[_from][msg.sender].add(_tokens);
    balances[_to] = balances[_to].add(_tokens);
    emit Transfer(_from, _to, _tokens);
    return true;
}
```

Figure 14. Securing `transferFrom` method instead of `approve` method can mitigate the attack by preventing more token transfer than allowed.

`red[_from][msg.sender]`). In other words, `transferred` variable is a life time variable that accumulates transferred tokens regardless of allowance changes. This token is implemented as TKNv3[12] on Rinkby network and passed compatibility checks by transferring tokens between standard wallets. In terms of gas consumption, `transferFrom` function needs at about 37% more gas than standard `transferFrom` implementation which is acceptable for having a secure ERC20 token.

5. Conclusion

While this paper is a deep dive into a specific issue with ERC20, it also illustrates a number of higher level lessons for blockchain developers. What are they?

References

[1] Ethereum. Ethereum project repository. https://github.com/ethereum, May 2014. [Online; accessed 10-Nov-2018].

[2] Fabian Vogelsteller and Vitalik Buterin. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, November 2015. [Online; accessed 2-Dec-2018].

[3] Mikhail Vladimirov. Attack vector on ERC20 API (approve/transferFrom methods) and suggested improvements. https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729, November 2016. [Online; accessed 18-Dec-2018].

[4] Tom Hale. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack #738. https://github.com/ethereum/EIPs/issues/738, October 2017. [Online; accessed 5-Dec-2018].

[5] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. *International Conference on Financial Cryptography and Data Security*, 2019.

[6] OpenZeppelin. openzeppelin-solidity. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol, December 2018. [Online; accessed 23-Dec-2018].

[7] ConsenSys. ConsenSys/Tokens. https://github.com/ConsenSys/Tokens/blob/fdf687c69d998266a95f15216b1955a4965a0a6d/contracts/eip20/EIP20.sol, April 2018. [Online; accessed 24-Dec-2018].

[8] Wikipedia. Compare-and-swap. https://en.wikipedia.org/wiki/Compare-and-swap, July 2018. [Online; accessed 10-Dec-2018].

[9] Friedhelm Victor and Bianca Katharina Lüders. Measuring ethereum-based erc20 token networks. In *International Conference on Financial Cryptography and Data Security*, 2019.

[10] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. *arXiv preprint arXiv:1811.11645*, 2018.

[11] Jordi Baylina, Danil Nemirovsky, and sophiii. minime/contracts/MiniMeToken.sol. https://github.com/Giveth/minime/blob/master/contracts/MiniMeToken.sol#L225, December 2017. [Online; accessed 23-Dec-2018].

[12] Peter Vessenes. MonolithDAO/token. https://github.com/MonolithDAO/token/blob/master/src/Token.sol, April 2017. [Online; accessed 23-Dec-2018].

[13] Enrique Chavez. StandardToken.sol. https://github.com/kindads/erc20-token/blob/40d796627a2edd6387bdeb9df71a8209367a7ee9/contracts/zeppelin-solidity/contracts/token/StandardToken.sol, March 2018. [Online; accessed 23-Dec-2018].

[14] Nate Welch. flygoing/BackwardsCompatibleApprove.sol. https://gist.github.com/flygoing/2956f0d3b5e662a44b83b8e4bec6cca6, February 2018. [Online; accessed 23-Dec-2018].

[15] outofgas. outofgas comment. https://github.com/ethereum/EIPs/issues/738#issuecomment-373935913, March 2018. [Online; accessed 25-Dec-2018].

[16] M. Vladimirov and D. Khovratovich. ERC20 API: An Attack Vector on Approve/TransferFrom Methods. https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit#heading=h.m9fhqynw2xvt, November 2016. [Online; accessed 25-Nov-2018].

[17] Ethereum Project. Create your own crypto-currency). https://www.ethereum.org/token, December 2017. [Online; accessed 01-Dec-2018].

[18] Dexaran. ERC223 token standard. https://github.com/ethereum/EIPs/issues/223, March 2017. [Online; accessed 12-Jan-2019].

[19] Steve Ellis. transferAndCall Token Standard. https://github.com/ethereum/EIPs/issues/677, July 2017. [Online; accessed 12-Jan-2019].

[20] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. ERC-721 Non-Fungible Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md, January 2018. [Online; accessed 12-Jan-2019].

[21] Jacques Dafflon, Jordi Baylina, and Thomas Shababi. EIP 777: A New Advanced Token Standard. https://eips.ethereum.org/EIPS/eip-777, November 2017. [Online; accessed 12-Jan-2019].

[22] Augusto Lemble. ERC827 Token Standard (ERC20 Extension). https://github.com/ethereum/eips/issues/827, January 2018. [Online; accessed 12-Jan-2019].

[23] Witek Radomski, Cooke Andrew, Philippe Castonguay, James Therien, and Eric Binet. ERC-1155 Multi Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1155.md, June 2018. [Online; accessed 12-Jan-2019].

[24] Atkins Chang, Noel Bao, Jack Chu, Leo Chou, and Darren Goh. Service-Friendly Token Standard. https://github.com/fstnetwork/EIPs/blob/master/EIPS/eip-1376.md, September 2018. [Online; accessed 12-Jan-2019].

12. https://rinkeby.etherscan.io/address/0x5d148c948c01e1a61e280c8b2ac39fd49ee6d9c6