Recurring Payments on EVM-based Platforms

Sergii Grybniak
Institute of Computer Systems
Odesa Polytechnic State University
Odesa, Ukraine
s.s.grybniak@op.edu.ua

Nicolae Goga
Molecular Dynamics Group
University of Groningen
Groningen, Netherlands
n.goga@rug.nl

Oleksandr Nashyvan
Institute of Computer Systems
Odesa Polytechnic State University
Odesa, Ukraine
o.nashyvan@op.edu.ua

Razvan Mihai
Faculty of Electronics, Telecommunications and Technology of Information (ETTI)
Politehnica University of Bucharest
Bucharest, Romania
razvan.mihai.phd@stud.etti.upb.ro

Igor Mazurok
Faculty of Mathematics, Physics, and Information Technologies
Odesa I.I. Mechnikov National University
Odesa, Ukraine
mazurok@onu.edu.ua

Yevhen Leonchyk
Faculty of Mathematics, Physics, and Information Technologies
Odesa I.I. Mechnikov National University
Odesa, Ukraine
leonchyk@onu.edu.ua

Gora Datta
Department of Civil and Environmental Engineering
University of California Berkeley
Berkeley, United States
gora.datta@berkeley.edu

Omer Faruk Ozkul
Faculty of Engineering in Foreign Languages (FILS)
Politehnica University of Bucharest
Bucharest, Romania
omer_faruk.ozkul@stud.fils.upb.ro

Constantin Viorel Marian
Faculty of Engineering in Foreign Languages (FILS)
Politehnica University of Bucharest
Bucharest, Romania
constantinvmarian@gmail.com

***Abstract*—This article proposes a concept of building a payment system for subscription-based services in cases of fixed prices and with recurring frequency. The proposed solution offers the user the opportunity for ease-of-use and transaction fee cost savings. The proposed method of calculating balances for crypto-wallets can be applied to other financial services that provide traditional banking products based on decentralized public platforms. The concept is deployed in the form of a smart contract for issuing tokens on Ethereum, which is backward compatible with ERC-20 and ERC-777, and can form the basis for a new EVM-based decentralized network standard.**

***Keywords*—recurring payment, token standard, smart contract, accrual accounting, subscription-based service, decentralized finance, blockchain.**

I. INTRODUCTION

Most modern computer-distributed systems have comprehensive infrastructures to support application functions [1]. In such systems, program codes are independently executed to a large extent, frequently on multiple network nodes [2]. Such decentralized applications (dapps) build on smart contracts that are part of the back-end and stored on a distributed ledger. Smart contract logic is implemented as a set of rules, recorded in open code (available to the public), that are enforced automatically, enhancing trust among participants. Cryptocurrency payments are transparent and secure, which makes them attractive for both customers and suppliers of goods and services [3].

Currently, the most popular system for implementing smart contacts is the Ethereum network, which provides for their execution via a unified Ethereum Virtual Machine (EVM) [4] and supports the object-oriented programming language, Solidity [5]. In addition, there is a number of platforms such as Fantom, Polygon, Tron, Avalanche, Waterfall, etc., which are compatible with the EVM.

Dapps have gained popularity in various social and business sectors due to their transparency, logic consistency, low market entry threshold, and low operating costs, relative to traditional services [6]. In addition, the suppliers are interested in accepting payments in cryptocurrencies in the interest of expanding their customer base to a large number of crypto-wallet holders. In exchange, consumers are given a wider choice of goods and services and the opportunity to save money by leveraging new payment systems.

The financial sector known as Decentralized Finance (DeFi) has experienced rapid growth. Investment amounts have increased rapidly, both in terms of the number of users, and the variety of services provided. New dapps have emerged recently, and the functionality of existing dapps has expanded [7], [8]. However, a number of acute problems have arisen, such as scalability, security, regulation, liquidity, usability, etc. All of these issues have inhibited the widespread use of dapps in the finance industry. A new generation of DeFi 2.0 protocols has emerged to resolve these problems, and to increase the comfort of using dapps (so-called user-centric design). This opens new possibilities for their mass adoption.

II. PROBLEM STATEMENT

In this work, we consider the following user story addressing recurring payments. A customer and crypto-wallet owner want to make automatic periodic payments, to ensure timely payments and save money. Regular payroll, insurance contributions, rental payments, and a host of other subscription-based services are highly profitable business scenarios for the proposed solution.

To do so, this customer must specify the supplier's account, the payment amount and frequency, and the contract start and end dates, with a prolongation option. Alternatively, the supplier of services may also play the contract initiator role. If during the subscription period there are not enough

https://doi.org/10.1109/iGETblockchain56591.2022.10087077

funds in the customer's account, an action procedure is required to initiate early termination of the contract.

III. Related Works

Cryptocurrency payments are based on the assumption that the initiator of the payment is always the owner of the wallet, because no one else can perform a transfer transaction from an individual account. Automatic payments, particularly recurring ones, are not natively supported by the most popular smart contract platforms. As a rule, the decision to transfer funds in such cases is based on the prior deposit of funds in the smart contract account, at the initiative of the crypto-wallet owner [9], [10]. In such custodial cases of holding funds, there is a technical possibility to freely dispose of those funds without the consent of their true owner. This allows for the building of a system of regular payments, or even payments on demand. In addition, a secondary market can be arranged where subscriptions can be bought and sold.

A non-custodial solution has been proposed in [11] and [12]. A customer immediately signs and sends a batch of transactions to a supplier, who stores it off-chain. Each billing period, the supplier signs and transmits one of these transactions to the Ethereum blockchain. The appropriate smart contract checks for validity and makes the payment. This does not offer any savings in transaction fees, but it does provide ease of use with fully automated crypto wallets. Despite the contract not being approved and the corresponding Ethereum Improvement Proposals (EIP-948 and EIP-1337) being closed, some other projects have developed their own solutions, based on this approach (e.g. [13], [14]).

Crypto exchanges (e.g. Binance [15], Crypto.com [16], etc) also offer automatic purchases of cryptocurrencies with credit cards, crypto, and fiat wallets. However, such solutions are centralized, and clients are deprived of the advantages that cryptocurrency provides – transparency and security.

Recent works [17], [18] have concluded that an entirely new token standard be developed using the accrual accounting method for convenient use and savings on transaction fees which could make recurring payments possible in a non-custodial manner. Such an approach can be seen as the next step in payment system development based on blockchain technology. However, the implementation of accrual accounting in decentralized public networks poses new challenges, since there is no mutual trust between participants. Hence, smart contract logic must properly describe all possible business scenarios and be resilient to various attacks.

IV. Token Standards

Dapps use a native (inner) token as a unit of account, designed to represent the balance of a digital asset [19]. Enterprise-class dapps have their own ad-hoc economics (tokenomics) to drive interactions between customers [20]. Therefore, smart contracts must be able to appropriately support the various scenarios and mechanisms needed for their implementation.

Tokens can be endowed with any properties implemented by the programming language in which the corresponding smart contract is written. However, to facilitate interactions between diverse smart contracts and make them compatible, a unified set of rules (functions or so-called standards), are introduced [21].

ERC-20. Currently, the most popular token standard is ERC-20, proposed in 2015 [22] and adopted in 2017 [23]. In particular, the standard allows Ethereum wallets to interact with one another by carrying out token transfer transactions, which has led to the emergence and rapid development of the "Initial Coin Offering" (a type of fundraising or crowdfunding) in the cryptocurrency industry [24]. ERC-20 also allows an account to give an allotment to another account, to enable the retrieval of a predetermined amount of tokens from it. However, with the current expansion of DeFi services, its feature set appears to be significantly limited, and does not allow a wide range of financial services to be fully implemented. In addition, ERC-20 has several inherent drawbacks and vulnerabilities [25].

ERC-777. The standard is backward compatible with ERC-20, but has a number of advantages [26]. In particular, an account can grant the right to send tokens on its behalf to other contracts or regular accounts. The standard also provides the ability to automatically cancel transactions with incompatible contracts and flag untrustworthy addresses. Token exchanges use one transaction instead of two, as in the ERC-20. Currently, ERC-777 is the most flexible standard, providing ample opportunities for use in the DeFi sector. However, there are still some scenarios that cannot be fully implemented.

V. Payment Logic Design

A. General Concept

The proposed payment system allows regular payments without blocking funds, which arise in the system in reaction to unsecured transfer transactions. Payment arrears may occur, because it is impossible to guarantee that sufficient funds are available in the account for the entire period when a regular payment is created and confirmed.

We will call a “short-term payment commitment” (and put into the debt queue) those payments for which the deadline is approaching, but where there are not enough tokens in the account. Information about credit histories is public and can be collected from Ledger data. The party interested in the payment has the opportunity to assess the payer's credit rating and decide its reaction to the payment. For example, to decide to provide services or transfer goods if the sender's credit rating is high enough. If the sender’s credit rating is low, the recipient of the payment may ignore the payment and choose not to provide goods or services.

In case of insufficient funds in the subscriber's account, a supplier follows the same logic when choosing between granting credit or terminating the contract. In the latter case, the supplier is obliged to send a termination transaction in order to terminate the contract. Otherwise, he will automatically default to granting credit, with the obligation to provide the agreed-upon goods or services. Note that each party has the right to terminate the contract at will at any time. A customer must also send a transaction to terminate the

subscription's validity and the corresponding payments for the next billing period.

B. Balance and Transfer Types

To solve the problem, for each address, the following elements are accounted for:

- ***final balance*** of made payments. Its value can be taken only without negative values. A payment is considered valid (even partially) if there are funds in the sender's account at the time of transaction execution.
- ***debt repayment queue*** is formed on the basis of funds transferred and calculation of the value of regular payments. If the payment allows for partial execution, only outstanding debts are taken into account.
- ***overdue debt*** to the account is the amount for which the deadline has already arrived, but has not yet been fully repaid due to insufficient funds in the payer's account.

According to the type of payment initiator, funds transactions are divided in the following ***three types***: transfer on the initiative of

1. a payer;
2. a recipient;
3. a third party who is neither the recipient nor the payer.

All of the above types of transfers have the following properties.

Acceptance. Type 3 transactions may be accepting or non-accepting to the beneficiary. Acceptance transactions require the recipient's mutual or prior permanent consent to receive funds in his account from the sender. Transactions in relation to the payer are always accepted. Transactions of type 1 are confirmed by the very act of sending with a signature, and do not require the consent of the recipient. Type 2 and 3 transactions require reciprocal consent from the payer to debit funds from his account. Confirmation for transactions of this type can be issued in advance on a permanent basis, but with an indication of the total amount of charges. In type 2 transactions, the recipient of funds is confirmed, and in type 3 transactions, their sender is confirmed.

Regularity. All listed transaction options can be both one-time and recurring payments. In addition to the amount and the addresses of parties for recurring payments, it is necessary to specify additional attributes such as time of the first payment, time of the end of payments, and their periodicity. A transaction is rejected (considered invalid), if by the time of forming a block with this transaction, the time of the first regular payment has expired. In other words, so-called backdated payments are forbidden.

When implementing a transfer transaction, three bit fields must be provided for setting divisibility, confirmability, and regularity flags.

C. Debt Processing

When funds appear in the account the debt queue is reviewed, starting with the oldest debt, regardless of its type, for repayment of debts formed earlier. Non-severable debts are charged only if there are enough funds on the account, and the severable debt can be repaid partially by the maximum possible value. Review of the queue stops when the incoming funds are exhausted. All debts that have not been charged remain in the queue and will be reviewed the next time funds are received.

We take a look at an example where transactions on Alice's account gradually pay off her debts. A capital letter indicates the payee and an asterisk* marks the separable transaction – in this case it is the first in the queue.

TABLE 1. AN EXAMPLE OF DEBT REPAYMENT

Debt queue	Transactions on Alice's account	Alice's account status
B*(20), C(100), B(1), C(2)	5	0
B*(15), C(100), B(1), C(2)	17	1
C(100), C(2)	50	49
C(100)	50	99
C(100)	10	9

D. Wire Transfer

The main problem with wire transfer payments is the need for each transaction or request for account status to run the entire chain of regular payments that have matured since the last call, before paying off the debts in the queue. Obviously, a smart contract cannot keep track of when the next regular payment is due, at the moment of funds wiring. Calling (execution of) a smart contract and all its work is done with any transfer of funds or balance request on an account. During the transaction procedure ***recalc***(id_{from}) is invoked for the payment sender. The ***recalc***(id) procedure updates balances, not only for the address specified in the argument, but also for all those linked to it by regular payments or debts.

The state of the account – $State(id)$ for each address id has information about the number of tokens account(id) and the debt queue, identified as $D(id) = Queue(d_i^{id})$. In this case, each regular payment to a certain address forms a separate debt. A series of non-severable debts to the same creditor are combined, while a series of severable debts is not.

Here is a step-by-step description of the ***recalc*** (id) algorithm:

1. If the procedure call is initiated by a funds transaction from the address $id = id_{from}$ to the address id_{to}, then we add this payment to the queue of debts $D(id) = D(id) \cup d_{tx}$ and create a queue to browse addresses $Plan = \{id_{to}\}$ and a list of addresses $\text{Neighbours} = \{id_{to}\}$ to recalculate the balance.
2. Consistently, we look through all transactions of regular payments in search of regular payments from the address id. If another payment is found, we add it to the debt queue in the order of increasing time stamp

$D(id) = D(id) \cup d_{tx}$. If the address id_{to} has not yet been added to the watch queue, then we add it there $Plan = Plan \cup \{id_{to}\}$ idto and expand the list of nodes $Neighbours = Neighbours \cup \{id_{to}\}$ idto to recalculate balances.

3. After all payments from the address id have been fully processed, we exclude this address from the queue $Plan = Plan \setminus \{\text{id}\}$.
4. If $Plan = \emptyset$, then we repeat step 2 for the first address in the queue, $id \leftarrow Plan$.
5. After the completion of breadth-first search (BFS) [27] of the payments from the address id, we run a similar reverse BFS algorithm to search for payments to id. Consistently, we look through all transactions of regular payments in search of regular payments to $id = id_{to}$. If the next payment is found, then as in step 2, we add the debt queue and the list of nodes $Neighbours = Neighbours \cup \{id_{from}\}$ to the recalculation of balances.
6. After completing the reverse BFS, we generate a list of nodes $Neighbours$ whose balances can have a reciprocal effect on the initial balance id, as well as update the queue of debts (payments that were not made and yet to be paid) from these addresses $D = \{D(x) | \forall x \in Neighbours\}$.
7. To all $x \in Neighbours$ we consistently review $D(x)$ and make payments as long as the number of tokens in the account allows it. Payments that have been made are removed from the debt queue. If there are insufficient funds for the next payment, we move on to the next address.
8. If at least one payment is made in step 7, repeat step 7.
9. If no payment could be made, the process is completed.

E. Clearing System

The smart contract implements a system of netting mutual debts to reduce the mass of tokens in circulation and the number of transactions for consistent repayment of mutual debts. The procedure of ***clearing***(id) is called in the body of the procedure ***recalc*** (id) to resolve mutual settlements blocked by insufficient funds.

Consider a case where a clearing system reduces the computational complexity of mutual settlement. For example, Alice owes Bob 100 tokens, and Bob owes Alice 100 or more tokens. Suppose that both accounts are empty, and neither payment can be made due to insufficient funds. If there is no offsetting of debts, even a single token received in one of the accounts will trigger a large number of transfers. So the token received by Alice will immediately go to Bob for partial repayment of the debt, and will then immediately return back to Alice. This circular token journey will be repeated 99 times until both debts are paid off – a counterproductive calculation.

A reduction in the necessary token supply is clearly visible in the case of large mutual debts. For example, Alice owes Bob 1000 tokens, and Bob owes Alice 1001 tokens. For the actual balance in this situation, having a single token in Alice's account is sufficient. However, 1001 tokens are required for the situation to be formally resolved. This requires the number of tokens in circulation to be greater than necessary. In addition, these extra tokens will eventually end up in either Alice's or Bob's account. Until this happens, the contract memory will be tied up with storing excess information. The clearing system will debit this kind of debt without conducting transfer operations.

The ***clearing***(id) procedure, with the help of a depth-first search (DFS) [27] from the node id, finds loops in the chains of linked transfers and reduces the amount of debt by the value of the smallest debt in the cycle. The smallest debt is removed because it is completely paid off. At that point, DFS (id) is repeated, to identify the new cycle, or to determine the impossibility of mutual debt charging.

VI. IMPLEMENTATION

The proposed payment and token accounting scheme, which allows for regular payments, is implemented as a smart contract running on the Ethereum network, or another system with a compatible virtual machine. Two smart contracts were developed for backward compatibility with ERC-20 and ERC-777 standards respectively. The tokens issued on their basis demonstrated the declared properties. The functionality of smart contracts is currently being expanded, and the algorithms are being optimized. The code and test results are in the public domain https://github.com/waterfall-foundation/recurring-payment-contract.

We consider, for example, the interface for the ERC-20 standard-based token smart contract. The following structure is used to store the necessary data:

```
struct RegularPayment {
   uint256 id;
   address from;
   address to;
   uint startTime;
   uint endTime;
   RegularPaymentInterval interval;
   uint256 amount;
   bool isApprovedFrom;
   bool isApprovedTo;
   bool autoProlongation;
   address creator;
}
```

Further, we describe the functions included in the interface. The basic function `createRegularPayment` for creating a recurring payment (even if it only happened once) looks like this:

```
function createRegularPayment(
   address from, address to,
   uint startTime, uint endTime,
   RegularPaymentInterval interval,
   uint256 amount, bool autoProlongation)
external returns (uint256 id)
```

The call arguments specify the payment participants, the time interval and frequency of payments, a fixed payment amount, and the possibility of automatic extension to the next time interval equal to the original one. If you want the recurring payments to span indefinitely, you should specify the maximum value for `uint` for the end-of-time period

parameter. The function returns Regular Payment `id` indicating whether the operation succeeded. It can emits the `CreatedRegularPayment` event:

```
event CreatedRegularPayment(
   uint256 id, address creator,
   address from, address to,
   uint startTime, uint endTime,
   RegularPaymentInterval interval,
   uint256 amount, bool autoProlongation)
```

This event emitted, when Regular Payment `id` is created by `creator`. The payer and/or recipient must confirm the payment if it was created by someone else. Without such confirmation, the payment will be ignored by the system. The function `approveRegularPayment` is used to confirm the payment:

```
function approveRegularPayment
   (uint256 id)
external returns (bool success)
```

The function returns a boolean value indicating whether the operation succeeded and can emits an `ApprovedRegularPayment` event:

```
event ApprovedRegularPayment(uint256 id,
address user)
```

The event emitted, when the Regular Payment `id` is approved by `user`. Both the payer and the payee can cancel further regular payments from a certain point in time. For this, the function `cancelRegularPayment` is used:

```
function cancelRegularPayment
   (uint256 id, uint endTime)
external returns (bool success)
```

The function call signals about cancel Regular Payment `id` by the message sender. The parameter `endTime` is last date, when Regular Payment will be work. If `endTime` may be zero in which case `endTime` will be now, but not before the block in which the corresponding transaction will be written. The function returns a boolean value indicating whether the operation succeeded and emits the `CanceledRegularPayment` event:

```
event CanceledRegularPayment
   uint256 id, uint endTime, address user)
```

This event emitted, when the Regular Payment `id` is planned to cancel by `user` in `endTime`. The interface also contains some functions for checking the status of accounts and regular payments. Returns all unpaid Regular Payments by `user`:

```
function checkRegularPaymentsByUser
   (address user)
external view returns (RegularPayment[]
memory)
```

Returns the Regular Payment by `id`:

```
function getRegularPayment
   (uint256 id)
external view returns (RegularPayment memory)
```

Returns all Regular Payments by the message sender:

```
function getMyRegularPayments()
external view returns (RegularPayment[]
memory)
```

Returns all Regular Payments by `user`:

```
function getRegularPaymentsByUser
   (address user)
external view returns (RegularPayment[]
memory)
```

Returns all Active Regular Payments by `user`:

```
function getMyActiveRegularPayments()
external view returns (RegularPayment[]
memory)
```

Returns all Active Regular Payments by a message sender:

```
function getActiveRegularPaymentsByUser
   (address user)
external view returns (RegularPayment[]
memory)
```

Returns the unpaid amount of Regular Payment by `id`:

```
function getRegularPaymentAmount
   (uint256 id)
external view returns (uint256 amount)
```

The main interface functions of the ERC-20 standard remained unchanged. This allows new tokens to be backward compatible with ERC-20 tokens, wallets and dapps that support them:

```
function balanceOf(address account)
external view returns (uint256);
function transferFrom
   (address from, address to, uint256 amount)
external returns (bool);
function transfer
   (address to, uint256 amount)
external returns (bool)
```

However, when implementing these functions, the smart contract is supposed to carry out lazy evaluation (call-by-need) [29]. Those all regular payments are made only at those moments when we need to know the number of tokens in the account. This allows you to save computing power in case of short periods or a large number of regular payments in not very active accounts.

For example, Alice subscribed to a service for 10 token payable weekly and had 100 tokens on her account at the start of the subscription. After three weeks, she checks her balance by calling the smart contract with **balanceOf** method, and the smart contract executes: $100 - (10 * 3) = 70$ tokens provided no other transfers to/from Alice's account. A node, which got Alice's call, uses network time synchronized between nodes. Further, after one more week (four weeks from Alice's subscription inception), Bob sends 5 tokens to Alice: $70 + 5 - 10 = 65$ tokens on the account. In this case, the current date is obtained from the timestamp of the block in which Bob's transaction was included.

VII. CONCLUSIONS

The proposed new type of token can be used to deploy regular and recurring payments, e.g., providing loans with the settlement of credit histories or creating other traditional banking products in a decentralized environment. The chief advantage of the proposed solution is that there is no need to manually make recurring payment every billing period whilst offering easy of use and ensuring the sum of all transaction fees is less than the sum of fees of regular transfers.

Despite the fact that the smart contract is implemented for EVM-based platforms, the algorithms described in the work could be applied in other systems supporting smart contracts.

In the future, smart contracts may add the functions of suspending subscriptions, accrue late fees on incurred debts, etc. The possibility of interaction with Ethereum oracles [28], which, for example, could recalculate the amount of payment based on a fiat currency exchange rate or give confirmation of work performed, is also of interest. Non-custodial and fully open decentralized solutions expand the scope of opportunities and services provided to users of cryptocurrency platforms, which are typical for conventional banking products in payment systems. At the same time, the important advantages of public decentralized networks, such as transparency and security, will not be lost.

ACKNOWLEDGMENT

The technical details contained in this paper have been developed and prepared based on regular technical discussion and exchanges between members of Politehnica University of Bucharest, IEEE, Odesa Polytechnic State University, Odesa I.I. Mechnikov National University, and University of California Berkeley.

REFERENCES

[1] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," Journal of Network and Computer Applications, 177, 2021.

[2] C. Antal, T. Cioara, I. Anghel, M. Antal, and I. Salomie, "Distributed ledger technology review and decentralized applications development guidelines," Future Internet, 13 (3), p. 62, 2021.

[3] J. Liu, W. Li, G. O. Karame, and N. Asokan, "Toward fairness of cryptocurrency payments," IEEE Security & Privacy, 16(3), pp 81-89, 2018.

[4] E. Hildenbrandt, et al., "KEVM: A complete formal semantics of the Ethereum virtual machine," IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204-217, 2018.

[5] C. Dannen, "Introducing Ethereum and solidity," Berkeley: Apress, vol. 1, 2017.

[6] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain‐based decentralized applications," Software: Practice and Experience, 51(10), pp. 2033-2050, 2021.

[7] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (DeFi)," arXiv preprint arXiv:2101.08778, 2021.

[8] R. Selkis, "A Messari report: Crypto Theses for 2022," Messari, p. 165, 2022.

[9] P. Merriam, "Ethereum alarm clock," 2018. [On-line]. Available: https://github.com/ethereum-alarm-clock/ethereum-alarm-clock.

[10] J. Lai, "EIP-4885: Subscription Token Standard for NFTs and Multi Tokens," 2022, [On-line]. Available: https://eips.ethereum.org/EIPS/eip-4885.

[11] K. Owocki, et al., "Recurring Subscription Models are a Good Thing and should be viable on Ethereum (Merit + Architecture ERC)," 2018. [On-line]. Available: https://github.com/ethereum/EIPs/issues/948.

[12] K. Owocki, et al., "EIP-1337: Subscriptions on the blockchain," 2018. [On-line]. Available: https://eips.ethereum.org/EIPS/eip-1337.

[13] S. Burke, "How Groundhog uses the Gas Station Network," 2019. [On-line]. Available: https://medium.com/groundhog-network/how-groundhog-uses-the-gas-station-network-9a2530c8b715.

[14] M. Aggarwal, "Replayable Transactions, EIP 1337 implementation on Tezos," 2020. [On-line]. Available: https://forum.tezosagora.org/t/replay-able-transactions-eip-1337-implementation-on-tezos/2248.

[15] Binance, "How to Use Recurring Buy," [On-line]. Available: https://www.binance.com/en/support/faq/3b628537b6314964bb08b5b22fab6c18.

[16] Crypto.com, "Recurring Buy – How does it work?" [On-line]. Available: https://help.crypto.com/en/articles/4170965-recurring-buy-how-does-it-work.

[17] R. Mihai, "Universal Contract on Blockchain," Economics of Financial Technology Conference, Edinburgh, 2022.

[18] R. Mihai, O. F. Ozkul, G. Datta, N. Goga, S. Grybniak, and C. V. Marian, "Blockchain-Enabled Economic Transactions: Recurring Financial Accruals and Payments," unpublished.

[19] M. Shirole, M. Darisi, and S. Bhirud, "Cryptocurrency token: An overview," IC-BCT 2019, pp. 133-140, 2020.

[20] S. Au and Th. Power, "Tokenomics: The Crypto Shift of Blockchains, ICOs, and Tokens," Packt Publishing Ltd, 2018.

[21] L. Lesavre, P. Varin, and D. Yaga, "Blockchain Networks: Token Design and Management Overview," Internal Report 8301, National Institute of Standards and Technology, 84 p., 2021. [On-line]. Available: https://doi.org/10.6028/NIST.IR.8301.

[22] F. Vogelsteller and V. Buterin, "ERC-20: Token Standard," 2015. [On-line]. Available: https://eips.ethereum.org/EIPS/eip-20.

[23] T. Hale, "Resolution on the EIP20 APIi Approve/TransferFom multiple withdrawal attack," 2017. [On-line]. Available: https://github.com/ethereum/EIPs/issues/738.

[24] P. Cuffe, "The role of the ERC-20 token standard in a financial revolution: the case of initial coin offerings," IEC-IEEE-KATS Academic Challenge, Busan, 2018.

[25] R. Rahimian, and J. Clark, "TokenHook: Secure ERC-20 smart contract," preprint arXiv:2107.02997, 2021.

[26] J. Dafflon, J. Baylina, and T. Shababi, "ERC-777: Token Standard," 2017. [On-line]. Available: https://eips.ethereum.org/EIPS/eip-777.

[27] D. C. Kozen, "Depth-first and breadth-first search," The design and analysis of algorithms, New York, NY: Springer, pp. 19-24, 1992.

[28] B. Liu, P. Szalachowski, and J.Zhou, "A first look into defi oracles," IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), pp. 39-48, 2021.

[29] P. Henderson and J. H. Morris Jr, "A lazy evaluator," Proceedings of the 3rd ACM SIGACT-SIGPLAN, pp. 95-103, 1976.