University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Fee Optimization Mechanism in Blockchain-based Payments for an Open Source Donation Platform

Michael Bucher
Zürich, Switzerland
Student ID: 13-929-369

Supervisor: Eder John Scheid, Dr. Guilherme Sperb Machado,
Prof. Dr. Thomas Bocek
Date of Submission: December 16, 2021

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: https://www.csg.uzh.ch/

Abstract

Open Source Software (OSS) is widely prevalent these days. Most software contains integrated OSS to some extent. Nevertheless, many OSS projects are not pursued and maintained sufficiently due to the lack of funding. Although a significant number of donation platforms exist to support open-sourced projects, it is cumbersome for developers to keep up with receiving enough financial reassurance to sustain an OSS project. FlatFeeStack is a project that aims to solve this problem by allowing sponsors to donate a flat fee and pay the developers transparently using a decentralized approach. However, previous research on such a donation platform showed that performing a payout with viable transaction fees is challenging. Hence, in this thesis, an improved design is proposed in comparison to related work and supported by the implementation of a Proof-of-Concept. The solution uses a signature mechanism in combination with a batched payout, preserving important properties such as transparency. Conducted evaluations on two different blockchains (*i.e.*, Ethereum and Neo N3) show strong indications that on-chain fees can significantly be reduced. When high transparency is desired, the proposed approach decreases the payout fees on Ethereum on average by 76.5% up to 99%. Utilizing Neo N3, the fees can be reduced further by up to 99%.

Open Source Software (OSS) ist heutzutage weitgehend verbreitet. Die meiste Software beinhaltet integrierte OSS in gewissem Umfang. Dennoch werden viele OSS Projekte aufgrund mangelnder Finanzierung nicht genügend weiterentwickelt und aufrechterhalten. Obwohl zahlreiche Spendenplattformen existieren, die Open Source Projekte unterstützen, ist es mühsam für Softwareentwickler genügend finanzielle Mittel zu sichern, um ein OSS Projekt zu unterhalten. FlatFeeStack ist ein Projekt mit dem Ziel, dieses Problem zu lösen. Es erlaubt Sponsoren, eine Pauschalgebühr zu spenden, und bezahlt die Softwareentwickler mithilfe eines dezentralisierten Ansatzes transparent aus. Jedoch haben bisherige Erkenntnisse bezüglich einer solchen Spendenplattform ergeben, dass eine solche Auszahlung mit einer realisierbaren Gebühr anspruchsvoll ist. In dieser These wird daher ein verbessertes Design vorgestellt und mit einer Implementation konzeptionell demonstriert. Die Lösung beinhaltet einen Signatur-Mechanismus in Kombination mit einer gebündelten Auszahlung, welche wichtige Eigenschaften, wie beispielsweise Transparenz, bewahrt. Durchgeführte Evaluationen auf zwei Blockchains (*d.h.* Ethereum und Neo N3) zeigen starke Indizien, dass on-chain Gebühren signifikant reduziert werden können. Wenn hohe Transparenz erwünscht ist, dann reduziert der vorgeschlagene Ansatz die Gebühren auf Ethereum im Durchschnitt um 76.5% bis zu 99%. Wenn Neo N3 zur Auszahlung benutzt wird, können die Gebühren nochmals zusätzlich um bis zu 99% reduziert werden.

Acknowledgments

First, I would like to thank everyone that supported me in any way during the time I was working on this thesis. I want to express my gratitude to Dr. Guilherme Sperb Machado and Prof. Dr. Thomas Bocek for the many valuable brainstorming sessions and their technical assistance. Their passion when technical topics are involved is inspiring. Further, I would like to thank Eder John Scheid for his guidance in structuring and writing this thesis. Finally, I would like to thank Prof. Dr. Burkhard Stiller for providing me the opportunity to write my Master's Thesis at the Communication Systems Group.

Contents

Chapter 1

Introduction

Software that is Open Source can be accessed by anyone to inspect, extend and utilize. Most Open Source Software (OSS) projects are bound to a license which has to be accepted when using the software. It restricts the user from applying the software only in the means of the license holder. Provided that an OSS project's license conditions are complied with, it can be integrated and reused in any project. OSS is widely prevalent nowadays as most projects contain integrated or reused OSS to some extent in a wide variety from libraries like Tensorflow [1], code style improvement tools like ESLint [2], or full-fledged programming languages like Rust [3].

Compared to OSS, closed source software does not share the source code with the open community. Such software is produced internally with only a limited number of developers having insight, while in OSS, the whole open community can access the source code. More visibility ultimately leads to more trust because anyone can verify what the software is processing in detail. Since the entire code base is public, this further allows the user to choose whatever build or version there was in the past and also the possibility to customize or adapt the software to any specific use case.

It was shown in [4] that most OSS projects are only sustained less than ten days. Disengaging a started OSS project may have various reasons, such as, *e.g.*, the lack of interest in the community, the lack of time by the developer, or just the lack of any support. However, [5] found that the main driver to disengage in a project is the lack of funding. Often it is hard to find a sponsor. Foundations usually request an extensive plan and an existing project base to accept a funding request. This effort out scales any smaller project so that only bigger projects will ever consider applying to a foundation.

As there are many platforms to apply for sponsorships and receive donations, OSS contributors are incentivized to sign up on multiple platforms to increase their visibility to potential sponsors. This also reflects on the sponsors, as they may have to create multiple accounts on different platforms as well to find and support the projects they would like to back. Naturally, different donation platforms use different payment providers, such that sponsors, as well as contributors, carry the burden of maintaining multiple payment accounts. Eventually, involved parties find themselves in a complicated process where they have to maintain unnecessary many platforms and services.

In past research, [6] proposed a payment flow for a donation platform that offers a fair and transparent distribution of funds to OSS developers. Contributors should be able to verify that actual donation funds are present. Therefore, a decentralized approach with the Blockchain (BC) technology was chosen. The payment to the contributor would eventually be issued utilizing cryptocurrencies to maintain transparency. It was found that the fees for the payout to the developer are not viable, and as the demand for distributed ledger (DL) solutions is driving prices higher, a scalable solution that optimizes fees is required. This thesis answers the following research question:

Is it possible to design a transparent donation payment mechanism while lowering the payout fees significantly?

In this thesis, a potential solution approach is presented and evaluated. This approach aims to maintain the transparency of the former approach by [6] and lower the costs of the payout to the developers. A scalable operation should be enabled, and further support the possibility to be adapted generic within different BCs. The favored approach is presented, which combines the fee optimizations of the BC-based payments while improving transparency for the developer. The design is developed and thoroughly tested in a Proof-of-Concept (PoC) implementation for the BCs Ethereum and Neo N3. It includes multiple options for developers on how to receive or withdraw earned funds. These options are evaluated in realistic scenarios and compared to each other. Further, the emerging fees are compared between the implementations for Ethereum and Neo N3.

Firstly, related work and background information is described and discussed. Current problems in the area of OSS are specified and exemplified, and involved concepts within the same context are shown. Then, the design of the solution approach is closely elucidated. Afterward, the implementation details of the involved services and smart contracts (SC) are closely described while also providing implementation-specific optimizations. Finally, in Chapter 5, the constructed SCs for Ethereum and Neo N3 are evaluated by comparing the payout options on both BCs, as well as comparing the proposed solution with the former approach by [6] that has been implemented on Ethereum.

Chapter 2

Background and Related Work

This chapter provides background information and discusses related work that has been conducted.

2.1 Background

In this section, background information that is essential for the content of this thesis is described.

2.1.1 Signatures and Asymmetric Encryption

Asymmetric encryption is an encryption process that makes use of a key pair to encrypt and decrypt messages [7]. A key pair consists of a private key (PK) and a public key (PubK). The PubK is derived from the PK, which is why it is called a pair. Retrieving the PubK from a PK is simple, while deriving the PK from the PubK is extremely computationally expensive. Therefore, the PubK can be visible to anyone without any security concerns. In order to securely transmit a message from one person to another, the recipient holds a PK and provides the sender with the corresponding PubK without bearing any risk.

As exemplified in Figure 2.1, George holds a PK, and Fred wants to send a private message to George. For Fred to encrypt the message, the PubK of George can be used on the message. The encrypted message can then be sent without bearing the risk of anyone being able to read it since only the PK belonging to the used PubK can decrypt it and retrieve the original message. As George receives the encrypted message, he can now use his PK to decrypt and read the original message. Generally, a message to be signed can be arbitrarily large. However, to sign larger messages, more computational power is required. Hence, usually, a one-way hashing algorithm like SHA-256 [8] is applied to reduce the size of the message.

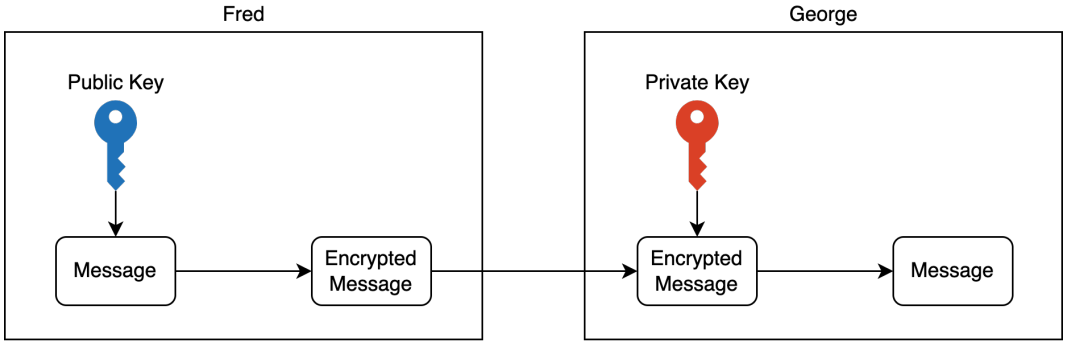

Figure 2.1: Asymmetric encryption process

When an encrypted message is received, the originator can not be verified. Anyone could have intercepted the transmission of the encrypted message between Fred and George and provided George with a fraudulent encrypted message since the PubK can be openly shared. Digital signatures [9] are used to verify the originator of a message. In order for George to verify that it was Fred that sent him the message, Fred can add a digital signature to the message. This message does not necessarily have to be encrypted since the signature process approaches a different issue. However, encryption and signature may also be used conjointly to keep a message private and ensure that the originator who encrypted that message can be verified. As the encryption of a message requires the recipient to hold a PK and share its corresponding PubK, the originator has to hold a key pair when creating a signature. Fred can use the PK on the message and then transmit the message together with the signature. George can then use Fred's PubK, the message, and the signature to derive the signing PubK. If Fred's PubK matches with the resulting PubK, the signature is valid, and George can be confident that Fred was the originator of the message.

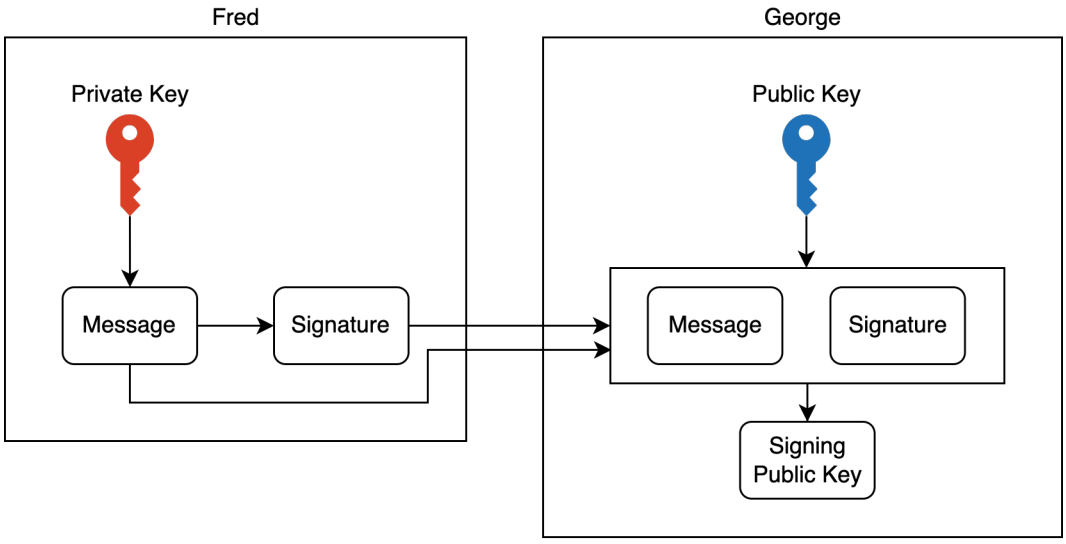

Figure 2.2: Signature process

2.1.2 Distributed Ledger Technology

A DL is a decentralized database that is maintained by multiple participating nodes. Updates to the database are issued through transactions. These transactions are required to follow specific rules that are defined in the protocol of the ledger. Depending on its protocol, either through voting or reaching a consensus between all or just a subset of all nodes, the update is accepted and included in the ledger. Since in a DL, the presence of malicious nodes is anticipated, it motivates robust consensus algorithms or stable voting restrictions to be adapted, such that transactions are ensured to follow the required rules strictly.

A BC is a particular case of a DL. It packs state updates in blocks where every new block that is created is dependent on the previous one, thus chained together. Since blocks are linked to the previous block, every future block is dependent on every previous block that was ever included in the chain. This provides high transparency due to the fact that not just the current state is public knowledge but also every change that was ever made to the ledger. Every block that is created consists of metadata called the block header, which includes the hash of the previous block and various additional information like a timestamp or its block number. A block further contains the accepted state changes referred to as transactions that have been issued to be included in the BC. Transactions are state updates executed on the BC to change its state.

Most BCs have a natively integrated fungible token as a digital asset that can be transferred within the BC. Such tokens are called cryptocurrency, whereas conventional paper money is referred to as fiat money. Usually, BCs use a fuel token that is spent to pay for issuing transactions, hence updating the state of the BC. The payment that has to be made is called the transaction fee and depending on different protocols, that fee is handled differently when paid. Usually, the fee is either burned, or distributed among one or multiple validators that verified the correctness of the issued transaction, or both, partially distributed and partially burned. In most BCs, there are additional fuel tokens minted and distributed every block. This is mandatory on BCs that burn parts of the fee to never run out of the token.

The participating entities in a BC are represented by addresses. An address holds a balance of the BC's token that allows it to interact with the BC and thus issue transactions. There are addresses that are derived from a PK and PubK pair (*i.e.*, externally owned accounts (EOA)), and addresses that belong to a Smart Contract (SC) (*i.e.*, accounts that are controlled by code).

Smart Contracts

A SC is a software program that is running on a DL and follows specified rules as to how to change the state of the ledger. Due to the fact that in a BC, every block is connected

and strictly dependent on the previous block, the algorithms of a SC are required to be deterministic as to derive the same state whenever executed again. On a BC, SCs can be embedded in different kinds of ways. The most common is as a script or on top of a virtual machine (VM) [10]. As the earliest BC, Bitcoin [11] is based on scripting, which allows executing simple programmable algorithms such as to transfer ownership of assets from one PubK to another. Later, [12] invented the concept of running SCs on top of a VM, which enables enormously greater possibilities for BC use cases. While scripting SCs can only execute simple stack-based operations, SCs that are implemented utilizing instructions of a VM can include far more complex algorithms as to how to handle and execute payments or borrow or lend tokens for interest, to name a few.

In the BC space, the terms *transaction* and *transfer* are largely used interchangeably. However, transactions on BCs are more than just a transfer of assets. Transactions contain additional information such as its hash, a timestamp, the block hash of the block it was included in, or signatures that allow verifying that assets may be spent. Further, a transaction also contains the script that should be executed on the BC. In VM-based BCs, this script holds instructions in the form of opcodes of that VM and additional bytes depending on what opcodes are used and what arguments this opcode requires to run on the VM. Thus, a transfer is merely a script to move assets from one account to another and is part of a transaction. In this thesis the terms *transaction* and *transfer* are used accordingly.

Ethereum

When the concept of running a VM integrated into a BC was introduced by [12], the Ethereum Virtual Machine (EVM) was born, which resulted in the development of the Ethereum BC. Ethereum allows the implementation of Turing-complete programs for the EVM utilizing the Solidity programming language [13]. It has a natively integrated token called Ether (ETH). ETH is divisible with 18 decimal places and is used to pay the emerging fees for transactions. The smallest unit of ETH is called *wei* and since there is a huge gap between one ETH and one *wei*, another widely used abstraction is called *gigawei* (*gwei*), which is equivalent to one billion *wei*. Further, Ethereum runs on a Proof of Work (PoW) consensus protocol, incentivizing network participants to build the next block by rewarding them with tokens for any new block added to the BC. These block builders are called miners.

In Ethereum, to execute a transaction, an amount of *gas* based on its script and size is required. Network participants can choose what maximal price they are willing to pay per *gas* and hence for their transaction. The fees are derived in an exclusively auction-based form where a maximum fee could be specified, and miners would choose the ones with the highest bids as the fees of all transactions in a block were rewarded to the miner. In August 2021, a controversial upgrade to the Ethereum protocol has been executed, referred to as the *London hard fork* [14]. It introduced a new concept of calculating the emerging fees of transactions defined in [15]. With the *London hard fork*, the Ethereum protocol introduced a base price per *gas* consumed by a transaction. The base fee is calculated based on the previous block's *gas* fee and a targeted value. To include a transaction, the issuer can now specify a maximum amount of *gwei* to pay for the base fee and add a tip

to it. When a transaction is included in a block, the base fee is burned, and the tip is rewarded to the miner.

Neo N3

The Neo BC [16], initially branded as AntShares, is a project that was founded in 2014. Recently, in August 2021, the latest update *Neo N3* was launched. Neo N3 is a VM-based BC that runs with a delegated Byzantine Fault Tolerance (dBFT) protocol [17]. The consensus is based on 7 validator nodes and 14 committee nodes that together form the Neo Council. It is responsible for running the network and can control various network settings with a single majority.

Neo N3 is based on a dual Token system with the native NEO and GAS tokens. NEO is an indivisible token that allows holders to participate in Neo N3's governance. NEO holders can vote for a node to get elected into the Neo Council, whereas one NEO counts as one vote. Anyone can issue a transaction to register as a candidate node. The 21 candidate nodes with the most votes form the Neo Council, out of which the top seven nodes are the validators that derive a consensus for each new block utilizing the dBFT protocol. GAS is Neo N3's utility token that allows taking part in using the BC. It is divisible with eight decimal places and is used for the emerging fees to interact with the BC. Every block, a number of GAS specified in the network settings is minted and distributed among NEO holders, voters, and council members.

Neo N3 features various natively integrated functionalities such as an oracle service, decentralized file storage, or a name service. Utilizing the dBFT protocol, Neo N3 supports one-block finality, which means that once a transaction is included in a block, the transaction will persist. Further, Neo N3 allows multi-language support, such that decentralized application developers can implement SCs and interact with N3 in commonly known programming languages such as Java, Python, Golang, and C# among others.

Other than Ethereum, Neo N3 uses a different approach to transaction fees. The network settings that the Neo Council can adjust consist of several parameters, such as a network fee per transaction byte or an execution fee factor that can define different kinds of fee factors. Basic transaction fees on Neo N3 consist of a system and a network fee. The system fee is based on the transaction script. Every opcode in the Neo N3 VM (NVM) is allocated a price, and the sum of all opcode prices multiplied with the execution fee factor specifies the system fee of a transaction. The network fee is derived by multiplying the network fee factor with the number of bytes the transaction consists of. Together, the system and network fee of a transaction form the total fee that has to be paid. An additional priority tip can be added to the network fee as the network fee is rewarded to the validator of the block while the system fee is burned.

2.2 Related Work

This section provides an overview of related work and discusses approaches to decrease transaction fees.

2.2.1 Approaches to Donation Platforms

Various solutions exist to donate to projects and other kinds of contributions. In the following, a few of them are elaborated on.

GitHub Sponsors

With a direct integration in the most prominent OSS supporting platform GitHub, GitHub Sponsors [18] is a well-known option to donate or apply for sponsorships. Both the benefactor as well as the developer to be sponsored are required to own a GitHub account and an additional GitHub Sponsors profile. As a benefactor, a one-time or monthly recurring payment can be made to sponsor a single developer or a full organization that applied and supports GitHub Sponsors to receive donations. As the project runs integrated with GitHub, the benefactor is required to own a GitHub account. Donations can be issued from a user account or from an organizational profile. The idea behind donations from organizational profiles is that developed projects in that organization can give something back to the developers of libraries or other OSS dependencies it is based on. The payment is issued through the global payment service Stripe Connect [19]. GitHub Sponsors itself does not currently charge a fee. However, it is noted that in the future, fees will be added to donations [20]. Currently, that means that just Stripe Connect may charge a fee between 1% and 2% in addition to potential fees depending on different account options.

Gitcoin

Gitcoin [21] is a project that pays developers to work on OSS in the Web3 space that is based on public BCs. Developers can earn cryptocurrency by tackling bounties on open GitHub issues, taking part in organized hackathons, or applying for grants. Gitcoin introduces quadratic funding, which increases the funding amount for a project substantially when more community participants support a project. Developers, as well as sponsors, have to own a GitHub profile and link it with the Gitcoin website. Sponsors can open issues and provide a bounty in the form of BC tokens for developers to solve them. Gitcoin charges 10% on top of that bounty as a platform fee, while developers that solve the issue are not charged any fee for the payout.

Open Collective

Open Collective [22] is a platform that offers transparent donations to various kinds of contributors, such as Open Source projects, clubs, or non-profit organizations. The sponsor can decide what project or which contributor to support and whether to issue one-time or recurring donations. The fee structure varies since there are multiple different fiscal hosts that can be chosen to issue a transfer of the donations to the beneficiary's bank account. The platform charges fees that span from 0% up to 30% with additional fees charged by the payment processor.

Flattr

Flattr [23] is a sponsoring platform that allows supporting developers, content creators, or projects. As Flattr is not bound to a strict platform where sponsorship applicants make contributions, sponsors and applicants for sponsorships are required to sign up and create a profile on Flattr. It provides the option of a one-time payment or a subscription. The sponsor can specify a payment amount and then choose the creators or projects to support. The chosen donation amount is then split evenly among all chosen beneficiaries. The payment is executed through the payment provider MangoPay [24]. Together with the fees charged by MangoPay, a donation on Flattr results in 10% fees, which leaves 90% of the total donation to be paid to the beneficiaries.

FlatFeeStack

Flatfeestack [25] is a project that aims to enable developers to make a living from OSS contributions. It introduces a flat fee donation approach that complies with the standard budget structures of companies and a transparent payout utilizing the DL technology. It supports version control management software, such as git. Sponsors can sign up for a flat fee subscription and then choose any and as many repositories as they intend to support. FlatFeeStack allows donations on projects rather than developers directly with the intent that companies can back the projects they are using. By sponsoring a project, they provide support for any future development of it. Other than most donation platforms, a donation to a project is only paid once further development has taken place.

In past research in the scope of FlatFeeStack, [26] developed a contributions analysis engine that can retrieve the commits of any public repository on GitHub, evaluate its commits of a specified time frame and provide a distribution of how much each developer contributed. Based on the results of the engine, donations issued to a repository are then split among the contributing developers. In order to maintain transparency, [6] found that with the use of SCs on Ethereum and Tezos, developers have to deal with a trade-off between trusting the donation platform and fewer fees for the payout transactions. As only direct payouts and less frequent transactions could lead to fewer fees, developers are required to fully trust the platform to execute the payment eventually. It was explicated that the payment process including funding the SC and the developer withdrawing the funds, a fee of about USD 1.08 emerged considering the ETH price trading at about USD 335 and the average gas price being approximately 60 *gwei* at the time.

Table 2.1 shows, that besides FlatFeeStack only Flattr provides the option of a flat donation. Considering the other platforms, sponsoring more projects ultimately leads to spending more funds on donations. Companies usually plan their budget in a periodical manner, usually quarterly or annually. For a company to consider a donation, it precedes careful evaluation of a project by the company as they would be required to specifically spend additional not planned expenses. This approach strongly favors already established projects compared to smaller projects, where the effort outweighs the matter. With a flat donation approach, companies can plan their budget accordingly and then change or add further projects to the sponsorship without budget concerns.

Table 2.1: Comparison of related donation platforms

	GitHub Sp.	**Gitcoin**	**Open Col.**	**Flattr**	**FlatFeeStack**
Flat donation	✗	✗	✗	✓	✓
Quality evaluation	✗	✗	✗	✗	✓
Transparent payments	✗	✓	✗	✗	✓
Payout fees	-	-	0-30%	10%	∼ USD 1.10

Further, only FlatFeeStack tackles an approach that includes an evaluation of contributions to fairly distribute donation funds in an OSS project. As for transparent payouts, FlatFeeStack aims to maintain transparency to the developer. However, with the current solution approach by [6], high transparency can only be delivered if the donations of a developer reach high monetary value. For developers that earn only small amounts of donations, the current solution is not viable as the transaction fees of the payout would diminish the paid amount. Hence, the funds would only be issued to the SC to be withdrawn as soon as it would become viable while there is no transparency up to that point. The fees for the payment to the developer do not seem too high for FlatFeeStack. However, it needs to be considered that at the time of the evaluation by [6], the market price of one ETH was about 7% of the current price, and with the grown demand in the network, the transaction fees have increased even more since then. GitHub Sponsors, as well as the Gitcoin platform, currently does not charge any fees utilizing the platform. However, it was mentioned that this might change for GitHub Sponsors in the future [20]. Further, Gitcoin charges a pay-in fee from the sponsor that is added on top of the sponsor amount. Furthermore, the payout fees tend to become higher for the other mentioned platforms compared to GitHub Sponsors, Gitcoin, and FlatFeeStack. Finally, while Open Collective, Flattr, and GitHub Sponsors only consider fiat money, the Gitcoin platform and FlatFeeStack utilize a payout with cryptocurrencies.

2.2.2 Takeaway on Fee Optimization

With the ever-growing demand in the BC space, the prices of cryptocurrencies have been growing enormously. As more people are using Ethereum, the network's capacity has been reached, which drives the fees to include a transaction in a block higher than ever. Transferring an ERC-20 token, as the technical standard on the Ethereum BC is called, costs around USD 26 at the time of writing [27, 28]. Invoking a complex SC method will derive an even higher fee eventually. Scaling solutions on Layer-1 are slowly implemented and integrated since it comes with risks that have to be assessed carefully, and ultimately a consensus in the community is required to fork the BC to upgrade the Layer-1 implementation. Thus, adapting the consensus to a less wasteful *Proof of Stake* or introducing *sharding* to increase scalability in the Layer-1 takes a long time.

In order to scale the use of BCs further, Layer-2 solutions are currently being developed with the aim to increase the transaction throughput and ultimately lower fees. Layer-2 solutions are implemented off-chain. Hence, they do not affect the implementation of Layer-1. The computational workload is done on Layer-2 while Layer-1 is utilized for verifying its correct execution. The most prominent approaches of Layer-2 solutions are

optimistic and zero-knowledge rollups, as well as plasma and state, or payment channels, respectively [29, 30, 31].

Channels are already widely established. The most prominent projects are the *Lightning Network* [32] for Bitcoin, and the *Raiden Network* [33] for Ethereum. In channels, a connection can be established in Layer-1 between two parties to exchange payments or allow state updates that concern both parties off-chain. A channel is opened on Layer-1 by locking the required token amounts by both parties in the channel. While the channel is open, the two parties can send tokens to each other off-chain or execute state changes. As soon as the channel is closed, the payments or state changes are updated on Layer-1, and the locked tokens are unlocked and distributed accordingly. While payment channels only support simple token transfers, state channels further allow agreeing on more complex state changes that are possible on SC supporting BCs.

In 2017, plasma was introduced by [34] as a framework that potentially could significantly increase the scalability of Ethereum. In plasma, sidechains are attached to another chain arranged in a tree structure, while the root chain is the Layer-1 of Ethereum. The proposed mechanism promised to vastly scale Ethereum. However, other problems arose under development. Each participant would have had to check the sidechains' state every, *e.g.*, two weeks to ensure no malicious state update was executed by the operator, a long period had to be waited to withdraw funds, and there turned out to be a problem when many participants would want to exit a sidechain in a short timeframe [34]. Subsequently, rollup approaches were introduced to solve the problems introduced by plasma while promising significant scalability improvements.

Within the last year, optimistic rollup projects have started to reach production. Two examples of optimistic rollups that are already running on Ethereum's Mainnet are the optimistic rollup chains *Optimistic Ethereum* [35], and *Arbitrum* [36]. In optimistic rollups, transactions are sent to an operator on Layer-2 that executes these and changes the state root accordingly on the Layer-2 chain. The state changes are then batched and *rolled-up* to the corresponding Layer-1 SC by storing data of these changes without executing transactions and thus not requiring further computational effort on Layer-1. Since with optimistic rollups, transactions are not actually executed on-chain, the *rolled-up* data's correctness may be challenged by issuing a *fraud proof* (*i.e.*, a claim that a batch is invalid). The Layer-1 SC specifies a timeframe (*e.g.*, one week) within which a *rolled-up* batch can be challenged. When a *fraud proof* is issued, it is verified on-chain in the rollup SC. In case it is valid, the *rolled-up* batch is reverted as well as every batch that descends of it. If a fraudulent *rolled-up* state change is not challenged by a *fraud proof* within the specified timeframe, the changes are final and can no longer be challenged. Thus, optimistic rollups require the participating parties to wait for the specified time for the certainty of executed changes.

In order to incentivize an operator to act truthfully, the operator is required to deposit an amount of ETH in the rollup SC. Additionally, participants of the Ethereum BC are incentivized to issue potential *fraud proofs* and hence pay for such a transaction on Layer-1 by rewarding the fraud prover with ETH tokens and returning a part of the emerged transaction fees. Consequently, in case a *fraud proof* is issued, and its correctness is

verified, the deposited ETH tokens of the operator are partly burned while another part is rewarded to the fraud prover [30].

Different to optimistic rollups, zero-knowledge rollups create a *validity proof* that is *rolled-up* to the rollup SC on Layer-1. A *validity proof* is a cryptographic proof that the reached state after the transactions of a batch have been executed is correct. Other than with optimistic rollups, where state changes are only final after there has not been any valid *fraud proof* been issued, *validity proofs* are verified whenever the state of Layer-2 is *rolled-up* to Layer-1. Thus, a batch that is *rolled-up* utilizing *validity proofs* is immediately final [37]. Few projects already provide zero-knowledge rollups on the Ethereum Mainnet (*e.g.*, the *Hermez* network [38], or the *Starknet* network [39]).

While most optimistic rollups are compatible with the EVM and Solidity, zero-knowledge rollups mainly support simple payments and are still in early development to support the EVM and potentially Solidity. With optimistic rollups supporting the EVM and Solidity, any transaction that may be executed on Layer-1 can also be executed on Layer-2 in the same way. Further, this allows designing, implementing, and optimizing an approach utilizing the tools of Layer-1.

Due to the fact that rollup solutions are not yet fully established, they have not been considered in the PoC implementation as it would add practical complexity that is out of the scope of this thesis. However, the proposed PoC implementation could be deployed and evaluated on Layer-2 solutions that support both the EVM and Solidity in the future. Nevertheless, adopting the proposed design on a Layer-2 rollup would ultimately optimize the fees on that Layer-2's fee level since the proposed design attempts to optimize fees on Layer-1.

Chapter 3

Design

This chapter presents the developed design of a fee optimized payout mechanism to OSS contributors. This mechanism has to meet specific requirements, which are described in the Section 3.1 before it is described in detail in the Sections 3.2, 3.3, and 3.4.

3.1 Context

On the FlatFeeStack platform, an end-user has to register and create a profile in order to interact with its services. Since a version control management system (*e.g.*, git) utilizes email addresses to identify the author of a commit, developers have to link their FlatFeeStack profile with the email address they use when contributing commits in that system. To issue a payout payment through a SC, they must provide and link a valid address of the BC they want to receive their payment. A FlatFeeStack profile can then be associated with contributions and a payout address.

For a sponsor to interact with FlatFeeStack, a profile has to be created, and a donation subscription can be chosen that specifies what amount is sponsored each month. Sponsors can then search for projects and star the ones that they would like to support. The timeframe in which a project is starred by a sponsor and the number of starred projects are later considered when evaluating which contributor earns how much.

In the former architecture of FlatFeeStack depicted in Figure 3.1, the analysis engine represents the commit evaluation engine introduced by [26]. The engine is triggered by the scheduler and evaluates the contributions of each project that is supported through FlatFeeStack. Whenever the payout is triggered by the scheduler, it queries the values resulting from the evaluation and calculates each developer's earned amount depending on the donations of the sponsors. The deployed SC holds a mapping that stores each address with its corresponding amount. The payout engine updates these values by issuing a transaction that includes the addresses and their calculated values. After this transaction has been executed, the developers can withdraw their earned funds from the SC to get paid for their contributions.

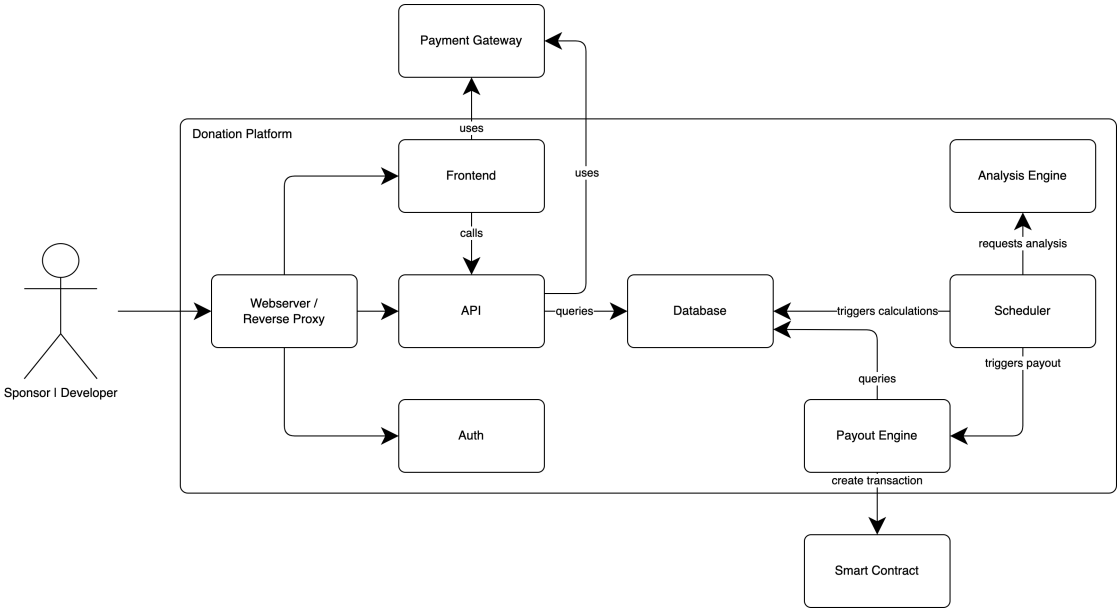

Figure 3.1: Former FlatFeeStack architecture [6]

The former payout design by [6] introduces multiple issues that reduce transparency and hinder its scalability. For example, transparency is only provided to the developer once FlatFeeStack updates the state of the SC. The funds are sent with the transaction of such an update. This means that the developers only see the funds once a transaction to update the SC's state has been executed. However, to show the developers beforehand that FlatFeeStack has funds at their disposal, the SC could be funded in advance (*e.g.*, at the beginning of every month). This would solve the problem with transparency to some extent. However, for a developer to verify that a specific amount of funds can be withdrawn, FlatFeeStack always has to execute a transaction first. The transparency of how much a developer has earned is therefore bound to the frequency that FlatFeeStack executes such transactions.

Consequently, [6] found that for small earned amounts of a developer, it is not viable to update the developer's balance in each transaction. The part of the transaction fees that emerges by including the developer would out scale the earnings. Thus, a developer with small earnings would have to wait for future earnings to reach the point where it would become viable to update the balance. This reduces transparency considerably. In addition, as soon as a balance update is executed eventually, the developer has transparency. However, another transaction is still required to withdraw those funds.

Since the former solution approach only provides transparency once an update of the SC has been issued by FlatFeeStack, the use of a *push and pull* payment through a SC is difficult to justify. The transfer of tokens could be integrated directly, which would reduce the summed-up cost eventually by lowering the number of required transactions. However, the scalability problem that transparency is bound to transactions persists. In order to release the tie to transactions, in this thesis' proposed approach, transparency is delivered by providing signatures off-chain.

The adapted FlatFeeStack architecture shown in Figure 3.2 illustrates the changes this

thesis introduces to optimize the payout while maintaining transparency. In order to integrate the changes into the current system, the former payout engine and the SCs are replaced, and a new component that provides signatures is included. The API needs minor adaptions, such as an added service to retrieve a signature, and the database requires updates based on the way a user's balance is tracked.

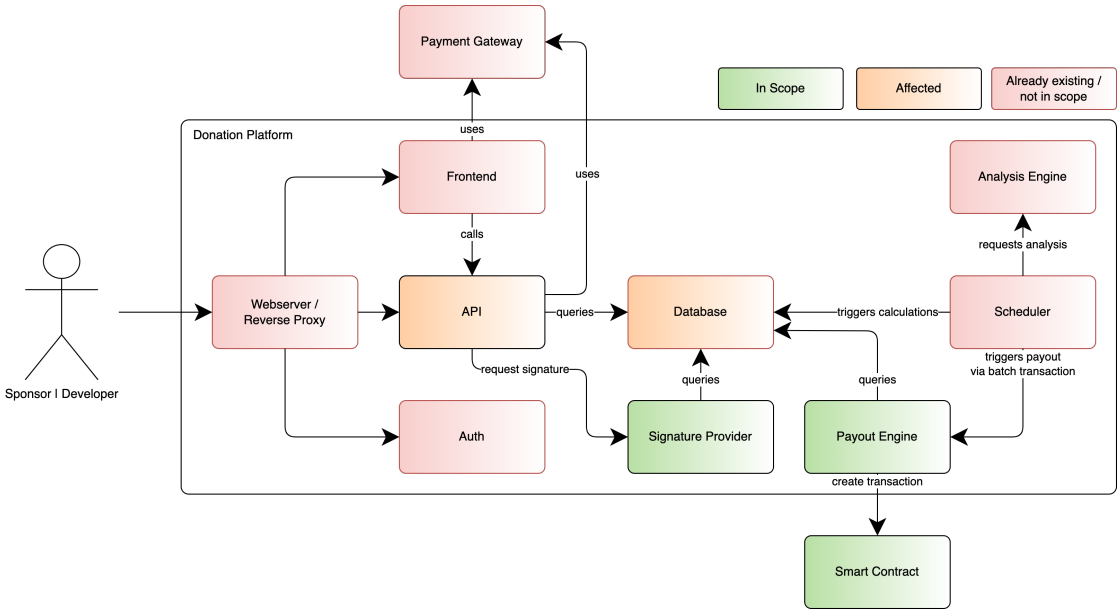

Figure 3.2: FlatFeeStack architecture

3.2 Signature Mechanism

One of the major aspects of this thesis is that the payout should be as transparent as possible, enabling developers to verify what balance they earned. The proposed signature mechanism solves this problem by providing developers with signatures to withdraw their current earnings.

Transferring assets from one address to another requires the owner of the sending address to witness such a transaction, *i.e.*, the owner has to sign the transaction so that it can be verified that the holder of the sending address has authorized that transfer. In case assets are held in a SC, the PK that derives the SC's address is not known. Therefore, no signature can be created, and no transaction can be signed without knowing the corresponding PK. However, SCs contain algorithms that can mathematically specify when a transfer is authorized to move assets away from the SC. One simple use case is that an EOA is stored as the owner of that SC, and whenever a transfer is executed, the algorithm checks whether its owner has witnessed that transaction. This way, the owner – and only the owner – is authorized to transfer funds away from that SC.

The proposed signature mechanism provides another algorithmic way of authorizing the withdrawal of assets from a SC. The designed SC holds a method that allows the withdrawal of assets without the SC owner signing a transaction. Developers can withdraw

earned funds by providing their address, their earned amount of funds, and a signature of the SC owner. This signature has to be created by the SC owner, while the message must match both the developer's address as well as the earned amount of funds. Since signatures are created by signing one single message, the bytes of the developer's address and the number of the earned funds are concatenated to form the message. It is then signed by the SC owner and provided to the developer. The developer can now use the corresponding address, the number of the earned funds, and the received signature as parameters to the SC's method and invoke it to withdraw the earned funds.

Within the method, the concatenated message is recreated from the provided parameters. The signer of the message and the signature parameter is then recovered and compared to the SC owner. In case the recovered signer and the SC owner match, the parameters are valid. The transfer is then authorized with the developer's address as the receiver and the provided amount of funds as the transfer amount.

As shown in Figure 3.3, the developer can request a signature through an API call. The returned response includes the current amount of earned funds and a signature that was created by the SC owner's PK. With that signature, the developer can issue a transaction on the SC to withdraw funds from the SC. The developer can withdraw the funds from the SC at any time in the future since the provided signature does not expire and remains valid.

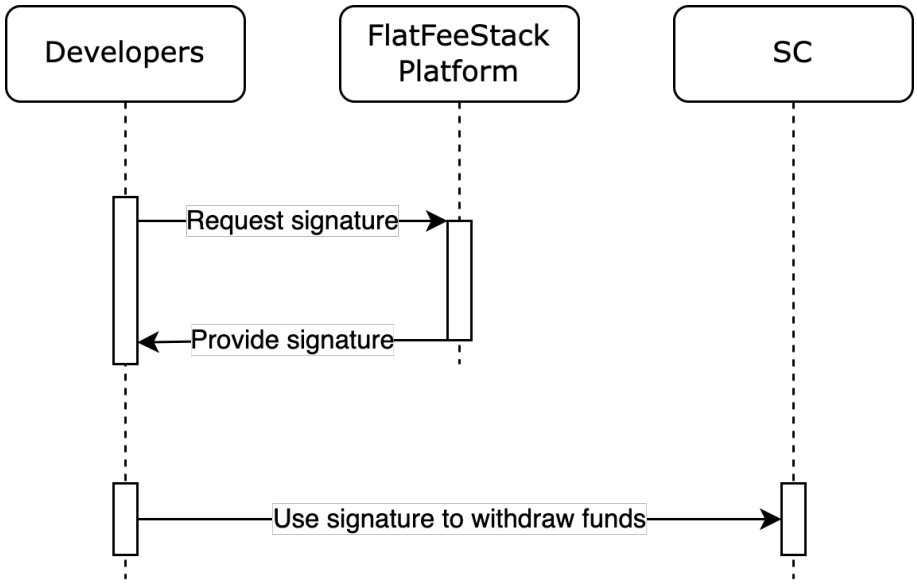

Figure 3.3: Signature mechanism

The signature provider component is responsible for creating signatures. Therefore, it requires access to the SC owner's PK. Since the PK is sensitive information, the component is detained in a decoupled separate service. This makes sure that only the signature provider component has access to the PK. Whenever a request for a signature is handled, the signature provider component receives the address and the number of earned funds and returns the bytes of the corresponding signature.

Developers can request a signature at any time. Hence, the received signature represents the current state of their amount of earned funds that is stored on the database of FlatFeeStack. Transparency is, therefore, provided synchronously with FlatFeeStack's database. Whenever the analysis engine runs and calculates how much every developer earned, the database is updated, and ultimately, any request for a signature will represent the current state of the database.

3.2.1 Signature Invalidation

Developers could reuse a signature to withdraw the same amount of funds again. In order to prevent this, signatures need to be invalidated once a withdrawal has taken place. Since signatures remain valid over time, they need to be invalidated indirectly within the SC so that funds can only be withdrawn once. Therefore, the number that is provided as the earned amount of funds is constantly increasing. Thus, it represents the total earned amount (*tea*) of a developer. Thus, when a developer earns funds, the newly earned amounts are added to the stored *tea* value in FlatFeeStack's database. Whenever a withdrawal takes place, this number is stored on the SC, and the transfer amount is calculated as the difference between the provided and the previously stored number on the SC. Any signature that represents a *tea* value that has already been withdrawn is hereby invalidated.

3.2.2 Pre-signed Transaction

Unlike on Ethereum, where only one sender is allowed to sign a transaction, on Neo N3, transactions can hold up to 16 signers that witness a transaction. These signers are ordered in a list where the first entry is the sender of the transaction and, therefore, is applicable to pay the emerging fees. This feature of Neo N3 allows authorizing additional functionality to be executed within one transaction script. Besides just providing a signature supporting Neo N3, FlatFeeStack can offer another option, which is a pre-signed transaction. Instead of requesting a signature, developers can request a pre-configured and signed transaction. This pre-signed transaction fulfills the same intent as the signature. Nevertheless, there are significant differences.

While a plain signature never expires, a pre-signed transaction contains a field *validUntilBlock* [40] that specifies a period in which a transaction is considered valid. As soon as the validity period has passed, the pre-signed transaction is invalid and no longer usable to withdraw funds. Due to the expiration of a pre-signed transaction, this option is intended to be used merely in case a withdrawal is wished to be executed in the near future. As of the time of writing, this period is approximately equal to three days. The main advantage of a pre-signed transaction is that it is already created and needs to be signed by the developer before executing it.

Further, the fees should still be paid by the developer, which means that FlatFeeStack's signer cannot be the first signer. This means that a signer is required to be set as first in the list, which is up to the developer to decide. Therefore, when requesting a pre-signed

transaction, the developer has to specify what address should pay for the fee. This address could be the developer's own address, as well as some other address that would sign that transaction to cover the transaction fees, such as a third-party service. This provided address will then be included as the first signer of the transaction, while FlatFeeStack will be put in the second index of the list.

Neo N3 transactions contain *signers* and *witnesses*. The signers are referred to as information about what address and with what scope the signature of this address may be used. A witness is the counterpart of a signer. It holds the signature data of the corresponding signer. The list of signers is included in the transaction data that is considered to be signed. Thus the signer list is final once FlatFeeStack pre-signs a transaction, which, after all, means that the first signer that pays for the fees of the transaction has to be fixed by the time it is signed.

3.2.3 Signature Workflow

Figure 3.4 shows the entire workflow with a provided signature or pre-signed transaction. FlatFeeStack funds the SC beforehand whenever donations are received in a simple transfer transaction. The developers contribute to the OSS platform continuously, and the FlatFeeStack backend collects the contributions periodically (*e.g.*, daily). These contributions are then evaluated with the analysis engine, and the *tea* on the database is updated for each developer according to the evaluated contributions and the donated values. The developer can then request the current state of the *tea* and further request a signature for this amount. This signature can then be held by the developer with the guarantee that the corresponding funds can be withdrawn at any time. Then, on the developer's initiative, a signature (*i.e.*, the signature for the highest *tea*) can be used to withdraw the funds from the SC.

3.3 Payout Engine

While the signature mechanism provides increased transparency to the FlatFeeStack platform, executing the payout in single transfer transactions still requires an additional transaction for each developer. In order to optimize the payout, a method in the SC is introduced that allows multiple developers to get paid in one single transaction. It takes as parameters a list of all developer's addresses to be included and a list of their corresponding *teas*. Further, it is restricted to the SC owner so that only the signature of the transaction needs to be verified to authorize the SC to transfer the corresponding funds. Hence, this batched payout is a service that is provided by FlatFeeStack for the developers since only FlatFeeStack as the SC owner can invoke this method. For FlatFeeStack to determine which developers to include in a batched payout, developers can subscribe to this service. Thus, developers are not automatically included.

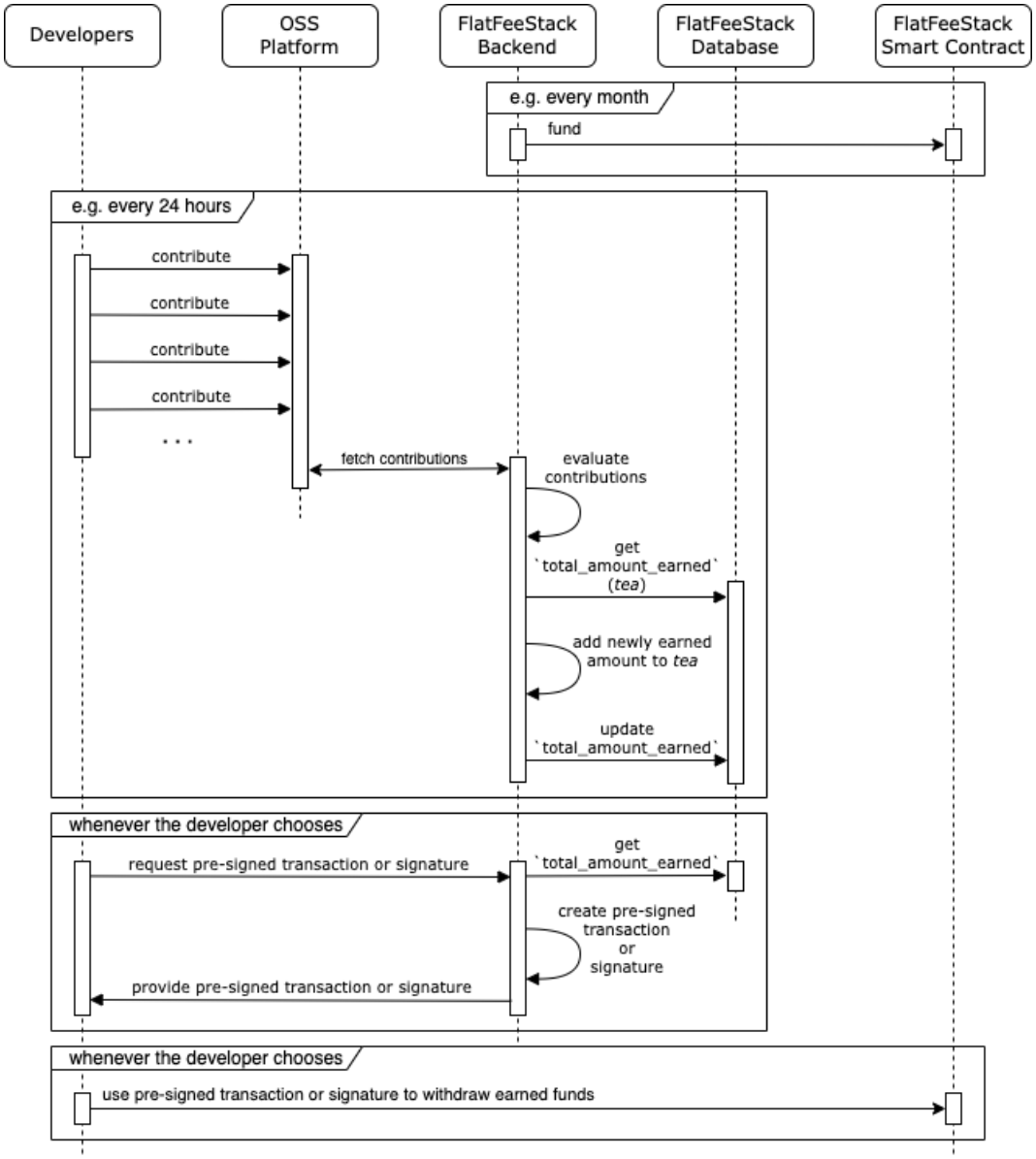

Figure 3.4: Signature workflow

The batched payout works similar to a withdrawal with a signature in terms of updating the SC with the stored *teas*. Thus, the signatures of the developers included in a batched payout are indirectly invalidated by updating their *tea* value in the SC alongside transferring the earned fund. Figure 3.5 shows the batched payout workflow. Developers can subscribe to the batched payout service (*e.g.*, by subscribing on the FlatFeeStack's website in their profile). Whenever FlatFeeStack plans to execute a batched payout (*e.g.*, every end of the month), the backend retrieves all subscribed addresses and their corresponding *teas*, builds the batch transaction, and then executes these transactions to pay the developers their earned tokens. Since there are limitations to the size of a transaction or a block, multiple transactions may be required depending on how many developers are considered to be paid.

3.3.1 Fee Charging Mechanism

The proposed batched payout functionality rises the issue that the transaction fees are paid by FlatFeeStack. As the batched payout is a service, the developers should be applicable to pay for their share of the emerging costs. Since the signatures can be received at any time and need to be invalidated, subtracting a service fee from the *tea* and use this reduced value for the payout is not eligible as it does not invalidate a potentially already provided signature by storing a lower value on the SC. An additional parameter referring to a service fee could be passed and subtracted from every *tea* value to consider the reduced value for calculating the payout amount. However, this increases the transaction fee due to a greater transaction size and additional calculations.

In order to charge a fee without the need to adopt a change on the SC and keep the algorithm on-chain as simple as possible, a fee charging mechanism is shown in Figure 3.6. The database requires an additional attribute called *include_next_batch* to check whether an account has already covered the service fee. In case the attribute value *include_next_batch* of that account is set to true, the service fee has been covered. Ultimately, if a batched payout transaction is created, this column is checked for all subscribed accounts, and only if it is set to true for an account, the account is included in the transaction since the service fee was paid. After the transaction is successful, these fields are set to false for all accounts that have been included in the batched payout transactions.

The service fee is paid indirectly by subtracting it from the *tea* when contributions are evaluated. As mentioned, it cannot be subtracted from a *tea* that has potentially already been included in an issued signature. Therefore, it is only subtracted whenever there is an update to the *tea* of an account in the database. Further, the update to the *tea* referred to as *newly earned amount* is required to be greater equals the service fee, since otherwise, the *tea* stored in the database would become smaller than a previously stored *tea* for which potentially already a signature was provided.

3.4 Smart Contract

The SC completes the proposed design. It is deployed on a BC and holds the necessary methods to fulfill the signature mechanism and the batched payout. The SC provides a method to withdraw funds with a signature and a method to pay out multiple addresses as described in the Sections 3.2 and 3.3. In order to retrieve information of the SC's storage, the SC offers methods to get an address's *tea* value or the SC owner. Additionally to these methods, the SC supports an exclusion functionality and an emergency fallback.

If a developer wants to change the receiving address, FlatFeeStack might not be able to verify whether a signature was provided for funds that have not been withdrawn. Hence, FlatFeeStack cannot provide a new signature for a different address without invalidating potentially created signatures for the former address beforehand. Therefore, the SC offers the functionality for the SC owner to update an address's *tea* to a greater value without

initiating a payment. By updating the *tea* value on the SC, any provided signature is indirectly invalidated and can no longer be used to withdraw funds from the SC. FlatFeeStack can now exclude this address from its database and reject developers from attaching it to their FlatFeeStack profile. After the update is performed on the SC, the developer can choose another address, and the *tea* value is adopted to the amount that was earned and had not been withdrawn yet.

Providing signatures is a sensitive task. In case there is a bug in FlatFeeStack's backend, signatures could be provided that match to *tea* values that are greater than the actual values. Developers who receive such a signature could then potentially withdraw funds from the SC that they have not earned. The SC, therefore, offers the functionality for the SC owner to change the SC owner as an emergency fallback. When the SC owner is changed, all provided signatures are immediately indirectly invalidated since the recovered signer of an existing signature will no longer match the SC owner. Thus, they can no longer be used to withdraw any funds. FlatFeeStack is still required to notice such a bug and act quickly to prevent the loss of funds.

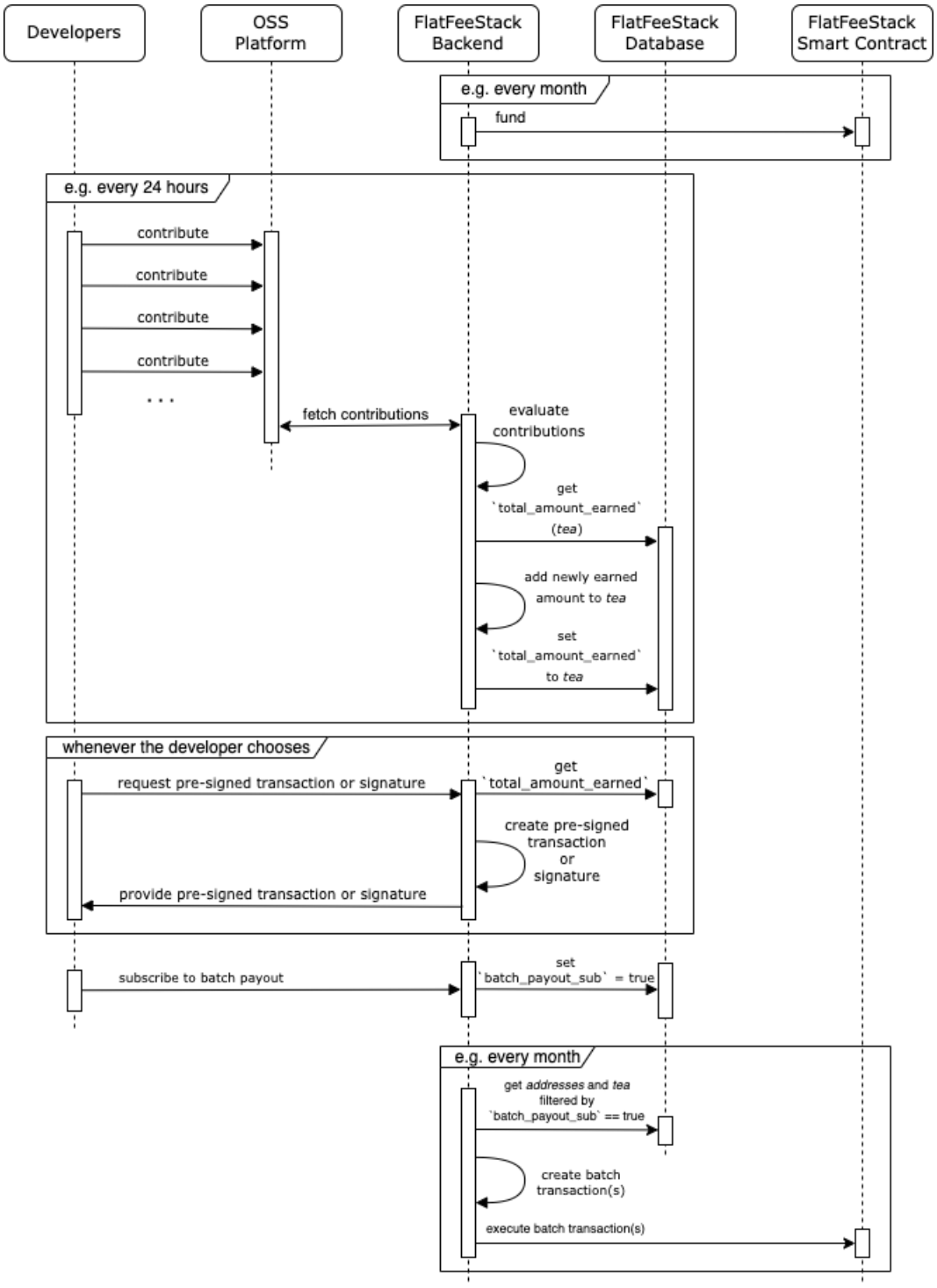

Figure 3.5: Batch payout workflow

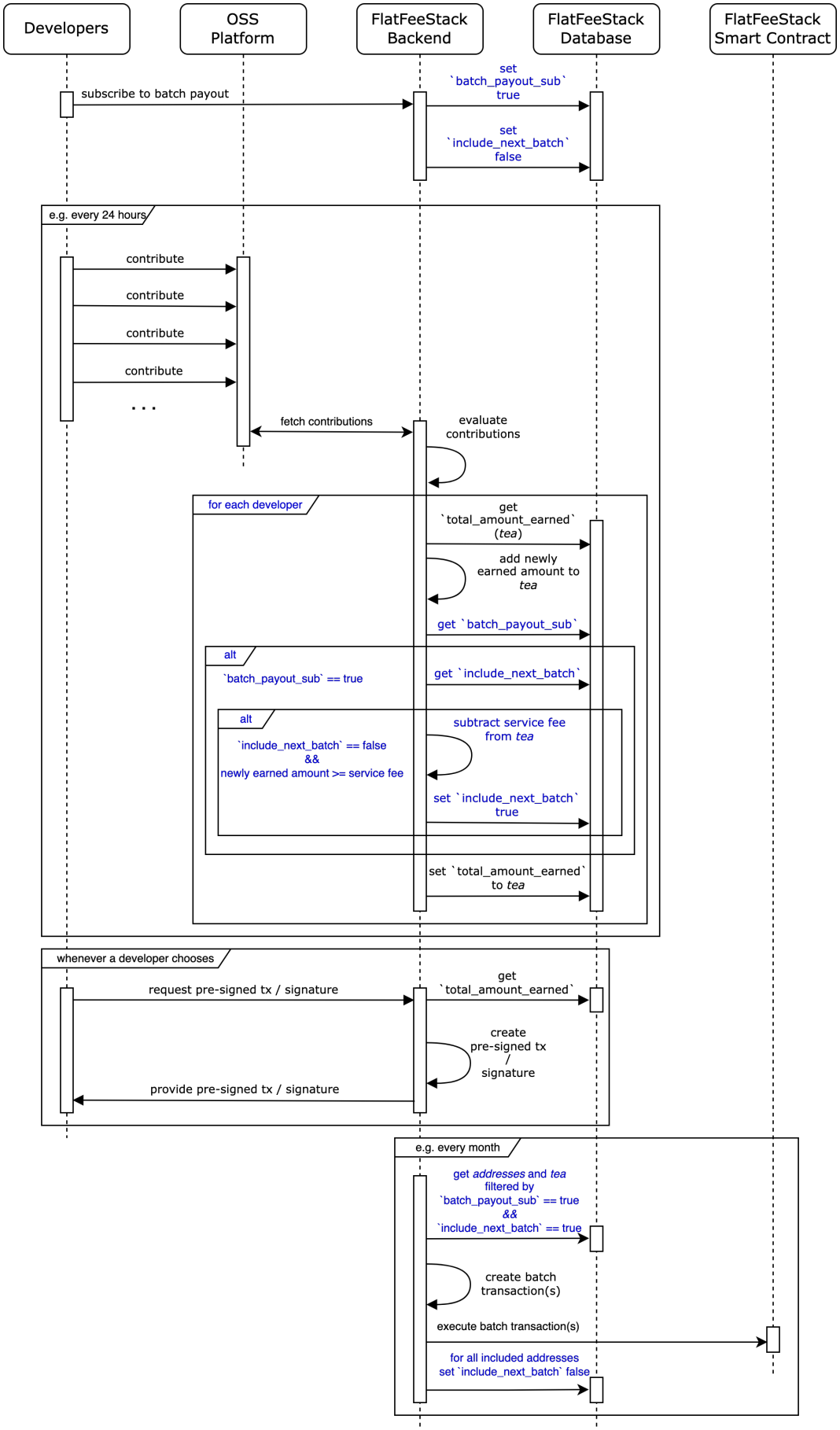

Figure 3.6: Service fee mechanism for batched payout

Chapter 4

Implementation

The changes to optimize the previous FlatFeeStack design described in Chapter 3 were implemented as a PoC. It consists of a SC for Neo N3, a SC for Ethereum, tests to verify the correct behavior of the algorithms programmed in the SCs, and a signature provider library with methods to generate signatures and deliver them to the FlatFeeStack's API service. The code is available at [41, 42, 43].

4.1 Signature Provider

The signature provider was implemented in Golang utilizing the `neo-go` Software Development Kit (SDK) for Neo N3 and `go-ethereum` for Ethereum, respectively [41]. The signature in Ethereum is different from the one created for Neo N3. This section elaborates on these different implementations.

4.1.1 Ethereum

Listing 4.1 shows how the message that is used for the signature verification on the SC is created for Ethereum. The address in hexadecimal is transformed to lowercase, and the *tea* in *wei* is padded to 32 bytes. These values are then concatenated, and its prefix `0x` is trimmed. These modifications are necessary since the concatenation of this message is executed on the SC where the bytes of the address are handled in lowercase, and the *tea* is handled as a 256-bit integer.

```
1 // Concatenates the address in lowercase with the tea padded
    ↪ to int64 and removes the prefix '0x'.
2 func getMessage(dev common.Address, tea *big.Int) string {
3   addressLowercase := strings.ToLower(dev.Hex())
4   paddedTea := padInt64(tea.Int64())
5   return strings.TrimPrefix(addressLowercase+paddedTea, "0x")
6 }
```

Listing 4.1: Concatenation of developer's address and *tea*

With the introduction of the EIP-712 signing standard [44], signed messages in Ethereum are prepended with a string of format `\x19Ethereum Signed Message:\n` and the length of the message. In Listing 4.2 the message is retrieved by the method `getHashedMessage` which gets the in Listing 4.1 declared message and hashes it. As on line 5, the hash's bytes are then prepended by the bytes of `\x19Ethereum Signed Message:\n66` before it is hashed again. These final hash bytes now match the signing standard and can be used to sign with the PK.

```
1 // Prepends the 'Ethereum Signed Message' string and message
    ↪ length to the hashed message and returns the hash of
    ↪ that.
2 func prepareForSigning(dev common.Address, tea *big.Int)
    ↪ []byte {
3   hashedMessage := getHashedMessage(dev, tea)
4   ethSignedMessage := []byte("\x19Ethereum Signed
      ↪ Message:\n66")
5   return ethcrypto.Keccak256(append(ethSignedMessage,
      ↪ hashedMessage...))
6 }
```

Listing 4.2: Adoption of EIP-712 signature scheme

The code presented in Listing 4.3 employs the use of the method introduced in Listing 4.2 and signs the prepared byte array utilizing the sign method of `go-ethereum`. The signature is now almost finished. However, the sign method of `go-ethereum` uses a recovery parameter of either 0 or 1, while they are required to be 27 or 28, respectively. These are changed accordingly, and then the signature is returned.

```
1  // NewSignatureEth Creates the hashed message from dev and
     ↪ tea and signs it with the provided private key.
2  // If changeRecoveryBit is true, the recovery bit is changed
     ↪ to 27 or 28, respectively.
3  func NewSignatureEth(dev common.Address, tea *big.Int,
     ↪ signingKey *ecdsa.PrivateKey, changeRecoveryBit bool)
     ↪ []byte {
4    preparedForSigning := prepareForSigning(dev, tea)
5    signature, _ := ethcrypto.Sign(preparedForSigning,
       ↪ signingKey)
6    if changeRecoveryBit {
7      // The Sign method sets the recovery parameters by
         ↪ default to 0 or 1, while in some cases they are
         ↪ required to
8      // be 27 or 28, respectively.
9      return setCorrectRecoveryParameter(signature)
10   }
11   return signature
12 }
```

Listing 4.3: Method `NewSignatureEth`

4.1.2 Neo N3

While Ethereum handles integers strictly as 256-bit values, Neo N3 aligns the needed bytes according to the passed integer value. Hence, no padding is required. Further, there also does not exist a signing standard that requires any prepending bytes. Listing 4.4 shows the message creation for Neo N3. The bytes of the developer's script hash (*i.e.*, another representation of an address) are concatenated with the bytes of the *tea* value.

```
1 // Concatenates the bytes of the dev public key with the
    ↪ bytes of the tea integer.
2 func prepareMessage(dev util.Uint160, tea *big.Int) []byte {
3   return append(dev.BytesBE(), bigint.ToBytes(tea)...)
4 }
```

Listing 4.4: Method `prepareMessage`

As shown in Listing 4.5, the prepared message is then passed to the method `Sign` provided by the `neo-go` SDK. There are no additional changes required. Thus, the returned byte array represents a signature that can be provided to the developer.

```
1 // NewSignatureNeo Signs the message created from the dev
    ↪ and tea with the provided private key.
2 func NewSignatureNeo(dev util.Uint160, tea *big.Int,
    ↪ signingKey *keys.PrivateKey) []byte {
3   message := prepareMessage(dev, tea)
4   return signingKey.Sign(message)
5 }
```

Listing 4.5: Method `NewSignatureNeo`

4.2 Smart Contract Functionalities

Table 4.1 shows an overview of the implemented SCs' functionalities. The purpose of each functionality is briefly explained. Further, it is shown whether the functionality is restricted to the SC owner (*i.e.*, the corresponding transaction has to be signed by the SC owner), or if anyone can issue a transaction to utilize it. Neo N3 supports all listed functionalities, while Ethereum offers a subset of it. The withdrawal with a pre-configured and pre-signed transaction is only applicable on Neo N3 since it requires multiple transaction signers.

4.2.1 Exclusion Functionality

The method `setTea` allows the SC owner to update the *tea* of an address without any payout. By setting the *tea* of an address higher or equal than the highest *tea* that was ever signed in a message for the corresponding address, all signatures that potentially have been provided for that address are thus invalidated without any payout for this address.

Besides excluding an address, this method further allows a developer to request to use another payout address. This could happen due to multiple different reasons, of which the most obvious would be that the developer lost the PK data. In case the currently stored *tea* on the SC equals the highest provided signed message, the address can be changed in the off-chain backend without any change on-chain. Otherwise, the SC owner needs to ensure that every signature provided for that address is invalidated before providing any signature for her new address since it cannot be verified that the PK data is lost. Thus, an update of the SC is necessary to assure this address cannot be used without any additional signature.

4.2.2 Emergency Fallback

In case of a bug in FlatFeeStack's backend, sensitive information could leak, or signatures may have been provided with *tea* values that are higher than the actual values. Thus, providing the opportunity for developers to withdraw more funds than earned. In order to invalidate all signatures that have ever been provided, the method `changeOwner` may be used to change the SC owner. Ultimately, the recovered signer of every signature provided leads to the former SC owner, which would no longer match the new SC owner's address in a withdrawal transaction and consequently would trigger a failed transaction. This action would invalidate both plain signature data and pre-signed transactions for Neo N3.

Table 4.1: Smart Contract functionalities

Functionality	**Description**	**Restricted to SC owner**	**Ethereum**	**Neo N3**
Withdraw with a signature	The developer receives a signature upon request that allows to autonomously release the payment at any point in time. Anyone may invoke the SC with this signature and hereby pay for the incurred fees.	✗	✓	✓
Withdraw with a pre-configured transaction that is signed by the SC owner	The developer receives a built transaction that is signed by the SC owner upon request. This enables the developer to autonomously release the payment at any point in time. The developer is required to pay for the emerging transaction fees since the address is included in the pre-configured transaction.	✗	✗	✓
Batch payout for subscribed developers	Pays out multiple accounts and updates their corresponding *teas*.	✓	✓	✓
Get *tea* of account	Gets the stored *tea* of an account.	✗	✓	✓
Set *tea* of account	Overwrites the stored *tea* of an account with a new value.	✓	✓	✓
Get SC owner	Gets the current SC owner.	✗	✓	✓
Change SC owner	Changes the owner of the SC.	✓	✓	✓
Fund SC	Allows the SC to receive assets.	✗	✓	✓

4.3 Ethereum

This section presents the implementations for the developed Ethereum SC [42]. The SC was developed in Solidity [13] and has been tested utilizing the Hardhat development environment [45].

4.3.1 Smart Contract

As shown in Listing 4.6, the SC holds a mapping `teaMap` that stores the developer's addresses and their corresponding paid out *teas*. Further, it holds the SC owner that is set when deploying the SC. Listing 4.7 shows the receive method that allows the SC to accept funds that are sent to the SC's address. Formerly, the method `fallback` was used for this. However, since there may arise security issues when including this method just for receiving funds, the method `receive` was introduced in Solidity 0.6.x to handle plain transfers to a SC [46].

```
1 mapping(address => uint256) public teaMap;
2
3 address public owner;
4
5 constructor () {
6     owner = msg.sender;
7 }
```

Listing 4.6: SC *tea* mapping

```
1 receive() external payable {
2 }
```

Listing 4.7: Method to receive funds

Listing 4.8 shows the withdraw method. It requires five parameters, which include the developer's address, the *tea* and the signature split into its three values. For a developer to withdraw funds, two requirements have to be fulfilled. The first is that the *tea* parameter is larger than the stored value for this address in the `teaMap`, and the second is that the signature matches with the address, the *tea*, and the SC owner. When checking the signature, the complete recovering of the signer is inlined to reduce transaction fees. First, the *tea* and the address are concatenated with `abi.encodePacked`. It is hashed with `keccak256`. Then, it is prepended according to Ethereum's signing standard, which is hashed again with `keccak256` before recovering the signer of this message and signature with `ecrecover`, and checking whether it matches the SC owner.

If these two requirements are met, the withdrawal is executed by first reading the old *tea* value, then updating the `teaMap` and transferring the difference between those two. The transfer is executed utilizing the `transfer` method. There is no need to check whether the transfer was successful since it throws an exception and reverts the transaction if it was not successful. The `transfer` method passes a fixed amount of *gas* to the recipient

in order to be able to execute potential additional instructions in the receiving method if the recipient is a SC. This could run into future problems in case that specific EVM instruction costs are changed. Ultimately, it is not recommended to use this method [47]. However, the use case in this thesis does not require transferring funds to a SC, and transferring funds to an EOA does not incur any additional *gas* fees as there are no additional instructions that could consume more *gas*. Therefore, it is assumed that the recipients are EOAs and not SCs.

In order for FlatFeeStack to verify that an address is an EOA and not a SC, a mechanism could be implemented in FlatFeeStack's backend to inspect any address that developers want to link as their payout address. For example, this implementation could request a signature from the developer that matches the provided address. If a signature can be provided, there is a strong guarantee that the developer holds the corresponding PK and the address is not a deployed SC.

Nevertheless, if payments should support SCs as recipients, Listing 4.9 shows an alternative approach to transfer funds according to [47]. Due to the mentioned reasons and since this replacement leads to a higher fee cost, it is not considered further in this thesis.

```
function withdraw(address payable _dev, uint256 _tea, uint8
  ↪ _v, bytes32 _r, bytes32 _s) public {
  require(_tea > teaMap[_dev], "These funds have already
    ↪ been withdrawn.");
  require(ecrecover(keccak256(abi.encodePacked("\x19Ethereum
    ↪ Signed Message:\n66",
    keccak256(abi.encodePacked(_dev, _tea)))), _v, _r, _s)
      ↪ == owner,
      "Signature does not match owner and provided
        ↪ parameters.");
  uint256 oldTea = teaMap[_dev];
  teaMap[_dev] = _tea;
  // transfer reverts transaction if not successful.
  _dev.transfer(_tea - oldTea);
}
```

Listing 4.8: Method to withdraw with a signature

```
(bool sent, bytes memory data) = dev.call{value: tea -
  ↪ oldTea}("");
require(sent, "Failed to send Ether");
```

Listing 4.9: Alternative implementation to transfer assets

Listing 4.10 shows the batched payout method. It is restricted to the SC owner with the modifier `onlyOwner` shown in Listing 4.11. The batched payout method consumes two lists, of which one holds all addresses of the developers and the other holds all *teas*. These are required to have the same length since every address is mapped to its corresponding *tea* value. The modifier `onlyOwner` is a simple check whether the message sender is equal to the SC owner to make sure only FlatFeeStack can execute this method. Then, in the loop that iterates through all entries in the address list, the corresponding developer, the

tea and the according `oldTea` that is in storage is fetched, and only if the provided *tea* is greater than the `oldTea`, the transfer is executed. Generally, it should never happen that the *tea* in storage would be greater. Otherwise, the backend of FlatFeeStack would have provided an incorrect value in a signature, or this method was invoked with an incorrect value, and the backend would need to be checked for other errors. While unlikely, it is still possible that a signature was provided, and while this batch transaction was set up and sent, a developer would have used the signature to withdraw with an equal *tea* value. In that case, a transfer of zero would lead to higher *gas* fees and is therefore ignored.

```
1 function batchPayout(address payable[] memory _devs,
    ↪ uint256[] memory _teas) public onlyOwner() {
2   require(_devs.length == _teas.length, "Arrays must have
      ↪ same length.");
3   for (uint256 i = 0; i < _devs.length; i++) {
4     address payable dev = _devs[i];
5     uint256 oldTea = teaMap[dev];
6     uint256 tea = _teas[i];
7     if (tea <= oldTea) {
8       continue;
9     }
10    teaMap[dev] = tea;
11    // transfer reverts transaction if not successful.
12    dev.transfer(tea - oldTea);
13  }
14 }
```

Listing 4.10: Method `batchPayout`

```
1 modifier onlyOwner() {
2     require(msg.sender == owner, "No authorization.");
3     _;
4 }
```

Listing 4.11: Modifier `onlyOwner`

Listing 4.12 shows the implementation of the method `setTea`. This method may be used by FlatFeeStack to update a *tea* value due to, *e.g.*, an address change. Besides the `newTea` value that should be set on the SC, the former value `oldTea` that is expected to be equal to the value in the storage, has to be passed in order to verify that no immediate withdrawal has taken place. This method is also provided with the same three parameters as lists to update multiple *teas* in one transaction. In order to ensure no transaction reversion in case an `oldTea` value is no longer correct, the update for the corresponding address is ignored as seen in Listing 4.13 on line 7.

```
1 function setTea(address _dev, uint256 oldTea, uint256
    ↪ newTea) public onlyOwner() {
2     require(oldTea == teaMap[_dev], "Stored tea is not equal
        ↪ to the provided oldTea.");
3     require(newTea > teaMap[_dev], "Cannot set a lower value
        ↪ due to security reasons.");
```

```
4     teaMap[_dev] = newTea;
5 }
```

Listing 4.12: Method to set *tea*

```
1 function setTeas(address[] calldata _devs, uint256[]
    ↪ calldata oldTeas, uint256[] calldata newTeas) public
    ↪ onlyOwner() {
2     require(_devs.length == newTeas.length, "Parameters must
        ↪ have same length.");
3     for (uint256 i = 0; i < _devs.length; i++) {
4         address dev = _devs[i];
5         uint256 storedTea = teaMap[dev];
6         uint256 newTea = newTeas[i];
7         if ((oldTeas[i] == storedTea) && (newTea >
            ↪ storedTea)) {
8             teaMap[dev] = newTea;
9         }
10    }
11 }
```

Listing 4.13: Method to set multiple *teas*

4.3.2 Implementation Alternatives

Alternative implementations have been developed and compared based on their *gas* cost. These alternative implementations were less optimized and thus have been superseded by the previously shown method variations.

Listing 4.14 shows an alternative for a withdraw method. Other than in the implementation shown in Listing 4.8, the signed message is not further hashed after concatenating the address and the *tea*. This results in a longer message to verify and ultimately a greater gas consumption. Further, considering reducing the size of the *tea* value using only 128-bit integers results in a higher cost, due to the fact that Ethereum matches this value to 256-bit, which requires an additional operation and ultimately does not reduce the message length.

```
1 function withdraw(address payable _dev, uint256 _tea, uint8
    ↪ _v, bytes32 _r, bytes32 _s) public {
2     require(_tea > teaMap[_dev], "These funds have already
        ↪ been withdrawn.");
3     require(ecrecover(keccak256(abi.encodePacked("\x19Ethereum
        ↪ Signed Message:\n106", abi.encodePacked(_dev,
        ↪ _tea))), _v, _r, _s) == owner, "Signature does not
        ↪ match owner and provided parameters.");
4     uint256 oldTea = teaMap[_dev];
5     teaMap[_dev] = _tea;
6     // transfer reverts transaction if not successful.
```

```
7       _dev.transfer(_tea - oldTea);
8   }
```

Listing 4.14: Withdraw method with unhashed message

In Listing 4.15 an alternative is shown for the batched payout. The proposed method includes charging a service fee to each developer. The provided *tea* is treated the same was as shown in Listing 4.10. However, the transfer value is deducted with the provided service fee, thus the developer pays the fee for FlatFeeStack's service with the earned tokens. This variation introduces an additional parameter and further arithmetic operations in each iteration, such that the *gas* consumption is strictly higher than the variation proposed in Listing 4.10.

```
1  function batchPayout(address payable[] memory _devs,
       ↪ uint256[] memory _teas, uint256 _serviceFee) public
       ↪ onlyOwner() {
2
3      require(_devs.length == _teas.length, "Arrays must have
           ↪ same length.");
4      for (uint256 i = 0; i < _devs.length; i++) {
5          address payable dev = _devs[i];
6          uint256 oldTea = teaMap[dev];
7          uint256 tea = _teas[i];
8          if (tea - _serviceFee <= oldTea) {
9              continue;
10         }
11         teaMap[dev] = tea;
12         // transfer reverts transaction if not successful.
13         dev.transfer(tea - oldTea - _serviceFee);
14     }
15 }
```

Listing 4.15: Batch payout method with service fee

4.4 Neo N3

This section shows implementation details for the developed Neo N3 SC [43]. The SC for Neo N3 was developed in neow3j [48] utilizing its devpack and compiler. The SC has been tested with a running Neo N3 private network [49].

4.4.1 Smart Contract

Each SC on Neo N3 has its own storage context which is retrieved with the method shown in Listing 4.16. This variable is static so that it is always loaded upon any invocation of the SC. The storage of a Neo N3 SC is based on key value pairs. Listing 4.18 shows how a

map is created. Listing 4.17 shows the `contractMap` which is used to store the SC owner. The `ownerKey` on line 2 represents the key to retrieve the SC owner.

```
1 static final StorageContext ctx =
    ↪ Storage.getStorageContext();
```

Listing 4.16: SC storage context

```
1 static final StorageMap contractMap = ctx.createMap(new
    ↪ byte[]{0x01});
2 static final byte[] ownerKey = toByteArray((byte) 0x02);
```

Listing 4.17: Storage key to store the SC owner

```
1 static final StorageMap teaMap = ctx.createMap(new
    ↪ byte[]{0x10});
```

Listing 4.18: Key prefix for `teaMap`

```
1 @OnDeployment
2 public static void deploy(Object data, boolean update) {
3   if (!update) {
4     ByteString pubKey = (ByteString) data;
5     // ECPoint instantiation checks valid public key length.
        ↪ Thus, makes sure that the data cannot be a Hash160.
6     assert checkWitness(new ECPoint(pubKey)) : "Passed
        ↪ public key must match a witness.";
7     contractMap.put(ownerKey, pubKey);
8   }
9 }
```

Listing 4.19: Deployment method

Neo N3 provides multiple native SCs that handle different aspects, *e.g.*, deployment, hashing, or executing an oracle request. The deployment of a Neo N3 SC is executed through an invocation to the native *ContractManagement* SC. Deploying a SC is done by invoking the `deploy` method of the *ContractManagement* SC with the information about the SC that should be deployed and arbitrary data as parameters. Within the deployment process, the *ContractManagement* SC initializes the new SC by invoking its `deploy` method with the provided data parameter. Listing 4.19 shows the deployment method of the FlatFeeStack SC. It contains the required parameter inputs and only executes instructions upon deployment as seen on line 3. The `deploy` method of a SC requires to have a boolean `update`, since SC on Neo N3 may also be updated through the *ContractManagement* SC, however, SC updates are not considered in this thesis.

The fungible token standard on Neo N3 is the NEP-17 standard [50]. If a SC should receive NEP-17 tokens, it is required to hold a method `onNep17Payment`. neow3j introduces this method with the annotation `OnNEP17Payment` as shown in Listing 4.20. It allows to transfer NEP-17 tokens directly to the SC address and does not require any other specific invocation on the SC. With this method included, the SC can also receive the native tokens NEO and GAS, since they both support the NEP-17 standard.

```
@OnNEP17Payment
public static void onNep17Payment(Hash160 from, int amount,
    ↪ Object data) {
}
```

Listing 4.20: Method to receive NEP-17 tokens

In Listing 4.21 the withdrawal with a signature is shown. It requires the script hash of the developer's account in the type `Hash160`, which is the type for 160-bit hashes in neow3j, the *tea* as an integer, and the signature data as a `ByteString`, which represents an immutable byte array. The method first verifies the signature with the provided values and the SC owner to ensure that the withdrawal is authorized. Then, the current value on the `teaMap` for the corresponding account is read, and the amount to withdraw is calculated. This value is required to be greater than zero since otherwise, this signature has already been used, and the funds have already been withdrawn. After these checks, the `teaMap` is updated with the new value, and the transfer is executed with the SC as the sender. In order to set the SC as the sender, the method `Runtime.getExecutingScriptHash` is used to retrieve its script hash.

The verification of the signature utilizes the native *CryptoLib* SC that supports the method `verifyWithECDsa`. This method verifies whether a signature, a message, and a PubK match. It requires an input of type `ECPoint`, which corresponds to a PubK. Thus, the FlatFeeStack SC requires to hold the PubK of the SC owner. Further, the `verifyWithECDsa` method requires the curve that was used to create the signature. The set value `23` corresponds to the elliptic curve *secp256r1*. An option to change this curve was dismissed due to the fact that changing it would invalidate existing signatures.

Due to the need of the SC owner's PubK, upon deployment shown in Listing 4.19, instead of the SC owner's script hash, the PubK has to be passed in the field of the arbitrary data, which is then set as the SC owner. The instantiation of an `ECPoint` is necessary since it checks whether the value's length equals to the length of a PubK, which is 33 bytes. An alternative would be to cast the data to an `ECPoint`. Casting the data does not check its length, and thus, passing a script hash would still pass the witness check, since `Runtime.checkWitness` accepts either a script hash or a PubK, and would thus successfully be deployed. However, the withdrawal method would no longer be usable since the initialization of an `ECPoint` would fail, and the signature verification would run into an error, resulting in a corrupted SC.

```
public static void withdraw(Hash160 account, int tea,
    ↪ ByteString signature) {
  assert CryptoLib.verifyWithECDsa(
    new ByteString(concat(account.toByteArray(),
      ↪ toByteArray(tea))), // the message
    new ECPoint(contractMap.get(ownerKey)), // the contract
      ↪ owner
    signature, // the signature
    (byte) 23 // the curve
  ) : "Signature invalid.";
```

```
8   int amountToWithdraw = tea -
      ↪ teaMap.get(account.toByteString()).toIntOrZero();
9   assert amountToWithdraw > 0 : "These funds have already
      ↪ been withdrawn.";
10  teaMap.put(account.toByteString(), tea);
11  assert GasToken.transfer(Runtime.getExecutingScriptHash(),
      ↪ account, amountToWithdraw, null) : "Transfer was not
      ↪ successful.";
12 }
```

Listing 4.21: Method `withdraw` with a signature

Listing 4.22 shows the withdrawal with a pre-signed transaction. Other than the withdrawal method in Listing 4.21, it runs a `Runtime.checkWitness` to check whether this transaction is authorized to be executed instead of a signature verification on the *CryptoLib* native SC. Similar to Ethereum, transactions on Neo N3 also hold one sender that signs the transaction and pays for the transaction fees. However, transactions on Neo N3 are further allowed to be signed by up to 16 signers in total. This allows checking the authorization of complex method invocations in a simple way. The signers of a transaction are included in the transaction data and thus are part of the transaction bytes. These bytes are signed by the specified signers, and signatures are then appended to the transaction. When all required signatures are present, the transaction can be sent.

The sender of a transaction is always the first signer in the list of transaction signers. Since the signers are included in the transaction bytes, this order is fixed once the transaction has been signed. The pre-configured withdrawal method requires FlatFeeStack's signature. However, FlatFeeStack does not want to pay transaction fees for the developer. Thus, when requesting a pre-configured withdrawal transaction, FlatFeeStack creates the transaction and orders the signers so that the developer's account is the first signer and FlatFeeStack is the second signer. That way, FlatFeeStack can witness the transaction without paying for the transaction fees itself.

```
1 public static void withdraw(Hash160 account, int tea) {
2   assert Runtime.checkWitness(new
      ↪ ECPoint(contractMap.get(ownerKey))) : "No
      ↪ authorization";
3   int amountToWithdraw = tea -
      ↪ teaMap.get(account.toByteString()).toIntOrZero();
4   assert amountToWithdraw > 0 : "These funds have already
      ↪ been withdrawn.";
5   teaMap.put(account.toByteString(), tea);
6   assert GasToken.transfer(getExecutingScriptHash(),
      ↪ account, amountToWithdraw, null) : "Transfer was not
      ↪ successful.";
7 }
```

Listing 4.22: Method `withdraw` with a pre-configured and signed transaction

Listing 4.23 shows the batched payout method. It requires two lists of the same length with the script hashes of the developers and their corresponding *teas*, and it is restricted

to be invoked by the SC owner. For each index of the lists, the stored *tea* is read, and the payout amount is calculated. In case it is greater than zero, the `teaMap` is updated, and the transfer is executed. Each transfer is required to execute successfully. Otherwise, the transaction returns a `Fault` state, which means that no change occurring from this transaction persisted.

```
public static void batchPayout(Hash160[] accounts, int[]
    ↪ teas) {
  assert checkWitness(new
      ↪ ECPoint(contractMap.get(ownerKey))) : "No
      ↪ authorization";
  int len = accounts.length;
  // Note: If teas had fewer items than accounts, the code
      ↪ would run into out of bounds anyways, but the other
  // way around that is not the case, thus this check is
      ↪ required.
  assert len == teas.length : "The parameters must have the
      ↪ same length.";
  for (int i = 0; i < len; i++) {
    Hash160 acc = accounts[i];
    int tea = teas[i];
    int payoutAmount = tea -
        ↪ teaMap.get(acc.toByteString()).toIntOrZero();
    if (payoutAmount <= 0) {
      continue;
    }
    teaMap.put(acc.toByteString(), tea);
    assert GasToken.transfer(getExecutingScriptHash(), acc,
        ↪ payoutAmount, null) : "Transfer not successful.";
  }
}
```

Listing 4.23: Batched payout method with list parameters

As well as the implementation for Ethereum, the Neo N3 SC contains a method `setTea` to update a *tea* value. Listing 4.24 shows the implementation of this method for Neo N3. It is restricted to the SC owner and the value `oldTea` must match the currently stored value to ensure no withdrawal in the meantime. The Neo N3 SC further supports a method `setTeas` to update multiple *tea* values in one transaction.

```
public static void setTea(Hash160 account, int oldTea, int
    ↪ newTea) {
    assert checkWitness(new
        ↪ ECPoint(contractMap.get(ownerKey))) : "No
        ↪ authorization.";
    int storedTea =
        ↪ teaMap.get(account.toByteString()).toIntOrZero();
    assert oldTea == storedTea : "Stored tea is not equal to
        ↪ the provided oldTea.";
```

```
5     assert newTea > storedTea : "The provided amount is
        ↪ lower than or equal to the stored tea.";
6     teaMap.put(account.toByteString(), newTea);
7 }
```

Listing 4.24: Method to set *teas*

In order to change the SC owner on the Neo N3 SC, the implementation shown in Listing 4.25 was developed. It requires a transaction that invokes this method to be signed by the current SC owner and the new SC owner. In this method, the `newOwner` does not require an additional check for a valid PubK length since the parameter type restricts that value from containing any invalid value. Thus, the `Runtime.witness` method with the PubK parameter is called and not the overloaded method that accepts a script hash.

```
1 public static void changeOwner(ECPoint newOwner) {
2     assert checkWitness(new
        ↪ ECPoint(contractMap.get(ownerKey))) : "No
        ↪ authorization";
3     assert checkWitness(newOwner) : "The new owner must
        ↪ witness this change.";
4     contractMap.put(ownerKey, newOwner.toByteString());
5 }
```

Listing 4.25: Method to change the SC owner

4.4.2 Implementation Alternatives

Different implementation variations have been developed and compared to each other. The following alternatives have been evaluated to emerge higher fee costs.

In Listing 4.26 an alternative method to the batched payout method in Listing 4.23 is shown. The NVM supports a map stack item that allows passing information in a map with key-value pairs. The method in Listing 4.26 makes use of that by passing the developer's addresses and their *teas* as key value pairs in a map parameter. Besides iterating through the key values of the map instead of a list's indexes, the methods are identical. However, the method supporting two list parameters consumes fewer fees and is thus favored.

```
1 public static void batchPayout(Map<Hash160, Integer>
    ↪ payoutMap) {
2   assert checkWitness(new
      ↪ ECPoint(contractMap.get(ownerKey))) : "No
      ↪ authorization.";
3   for (Hash160 acc : payoutMap.keys()) {
4     int tea = payoutMap.get(acc);
5     int payoutAmount = tea -
        ↪ teaMap.get(acc.toByteString()).toIntOrZero();
6     if (payoutAmount <= 0) {
```

```
7           continue;
8       }
9       teaMap.put(acc.toByteString(), tea);
10      assert GasToken.transfer(getExecutingScriptHash(), acc,
            ↪ payoutAmount, null) : "Transfer not successful.";
11  }
12 }
```

Listing 4.26: Batched payout method with a map parameter

Other alternative methods have been compared for the batched payout, that *e.g.*, included an additional parameter to deduct a service fee. However, these methods result in higher fees, and since a service fee can be charged off-chain as shown in Section 3.3.1, these variations have not been considered for the final SC.

4.4.3 Implementation Optimization

On Neo N3, no value might belong to the provided storage key when retrieving a value. In that case the returned value is `null`. In order to calculate the payout amount for a developer that withdraws or is paid out for the first time, the corresponding *tea* value is `null`. Therefore, a `null` check is required. This can be implemented in different variations.

In Listing 4.27, the *tea* value is preset to zero in case the value in the SC storage is `null`. If it is not `null`, it is converted within the `if` statement on line 4. Without presetting the *tea* value to zero, an `if..else` statement can achieve the same outcome while first handling a value being `null` (*cf.* Listing 4.28), or first handling a value that is not `null` (*cf.* Listing 4.29). Another alternative implementation `toIntOrZero` was developed in neow3j that circumvents a presetting or a `if..else` statement by either returning zero in case the value does not exist or an integer if it is present [51]. This method reduces the logic of the previously mentioned implementations to one single line (*cf.* Listing 4.30).

```
1 ByteString teaByteString = teaMap.get(account);
2 int tea = 0;
3 if (teaByteString != null) {
4     tea = teaByteString.toInt();
5 }
```

Listing 4.27: Method `toInt` (preset value to 0 with `if != null` check)

```
1 ByteString teaByteString = teaMap.get(account);
2 int tea;
3 if (teaByteString == null) {
4     tea = 0;
5 } else {
6     tea = teaByteString.toInt();
7 }
```

Listing 4.28: Method `toInt` (`if..else` with `== null` check)

```
ByteString teaByteString = teaMap.get(account);
int tea;
if (teaByteString != null) {
    tea = teaByteString.toInt();
} else {
    tea = 0;
}
```

Listing 4.29: Method `toInt` (`if..else` with `!= null` check)

```
int tea = teaMap.get(account).toIntOrZero();
```

Listing 4.30: Method `toIntOrZero`

In order to decide which of these implementation variations to use, the emerging GAS costs have been compared based on each implementation's fee consumption when executed integrated in a `withdraw` method. As shown in Table 4.2, the implementation of Listing 4.27 and 4.29 derive in a slightly lower fee consumption when no value exists on the `teaMap` for an account. However, as Table 4.2 shows, using the method `toIntOrZero` reduces the gas consumption to a greater extent. Considering scalability, an account that is paid a second time already compensates the first payout of five other accounts that have only been paid once, and further reduces the overall cost in each future withdrawal. Hence, the introduction of `toIntOrZero` not only reduces the implementation size but also decreases the fee consumption when considering paying accounts more than once.

Table 4.2: Comparison of `toIntOrZero` with non-existing storage value

Implementation option	**GAS cost (fractions)**	**Difference to** `toIntOrZero`
`toIntOrZero`	1816152	0
`if..else` with `== null`	1816152	0
`if..else` with `!= null`	1816146	-6
Preset to 0 with `if != null`	1816146	-6

Table 4.3: Comparison of `toIntOrZero` with existing storage value

Implementation option	**GAS cost (fractions)**	**Difference to** `toIntOrZero`
`toIntOrZero`	1630713	0
`if..else with == null`	1630743	+30
`if..else` with `!= null`	1630749	+36
Preset to 0 with `if != null`	1630752	+39

Chapter 5

Evaluation and Discussion

This chapter evaluates and discusses the design and implementation shown in the previous chapters.

In Section 5.1 the different payout options of the proposed approach are compared to each other based on their cost. Section 5.2 shows the evaluation of the former approach by [6] in order to provide necessary data for the comparison of use case scenarios that are obtained in Section 5.3. Finally, in Section 5.4 the results are discussed.

5.1 Fee Comparison of Payout Options

Cryptocurrencies tend to have high volatility. In 2021, the price of ETH fluctuated between a low of USD 736 in January to USD 4843 in October [52]. The *gas* price on Ethereum also fluctuates considerably. From October to December 2021, the lowest daily average was 50.96 *gwei* per *gas*, while the highest was 198.02 *gwei* per *gas* [28]. Similar to the fluctuation of Ethereum related prices, the prices in the Neo N3 ecosystem have been volatile as well. GAS tokens have been traded at a low of USD 1.50 in January 2021 up to a high of USD 19.79 in April 2021 [52]. For the following evaluation, approximate averages derived from the last three months have been considered (*cf.* Table 5.1).

Table 5.1: Evaluation factors

Factor	**Value**
Ethereum *gas* price	100 *gwei*
ETH price	USD 4300
Neo N3 GAS price	USD 6

The cost evaluation of the Ethereum SCs (*i.e.*, the SCs of the proposed and the former approach by [6]) has been conducted utilizing Hardhat's development environment [45] with a built-in local Ethereum network node. The *gas* consumption of each transaction was estimated using the *london hard fork* version of Ethereum. After each *gas* estimation, the corresponding transaction was executed, and its correct state change was verified to

ensure the correctness of the estimated values. The derived *gas* price values have then been multiplied with the Ethereum *gas* price and the price of one *gwei* according to the prices shown in Table 5.1.

For the cost calculations of Neo N3 transactions, the local private network `neo3-private-net-docker` [49] was used. In the same way, as with the evaluation of Ethereum transactions, each transaction fee calculation was followed by an execution of the transaction and a verification of the correct state changes. Transaction fees on Neo N3 consist of a *networkfee* and a *systemfee*. While the *networkfee* is based on the byte size of a transaction, the *systemfee* is based on the execution complexity of the transaction script. The following evaluation concerning Neo N3 transactions considers only the full transaction fee that is equal to the sum of the *networkfee* and the *systemfee*.

Figure 5.1 depicts the costs of the batched payout in Ethereum using the method `batchPayout` with accounts that are paid out for the first time, *i.e.*, no SC storage has been used by these accounts. The method was executed in order to pay different amounts of accounts. A maximum of 512 accounts can be paid since the execution of such a transaction reaches the maximal *gas* that is allowed per block. Thus, an entire block on Ethereum would be needed to include such a transaction.

The fees per account decrease when increasing the included accounts from 1 to approximately 20 accounts. While the fee per account is around USD 37, if only one developer is paid, the fee per account decreases to less than USD 26 for about 20 developers. The fee per account only decreases marginally when including more developers in the batched payout. With 50 included accounts, the fee per account costs USD 25.34, while it costs USD 25.12 for the maximum size of a batched payout with 512 accounts.

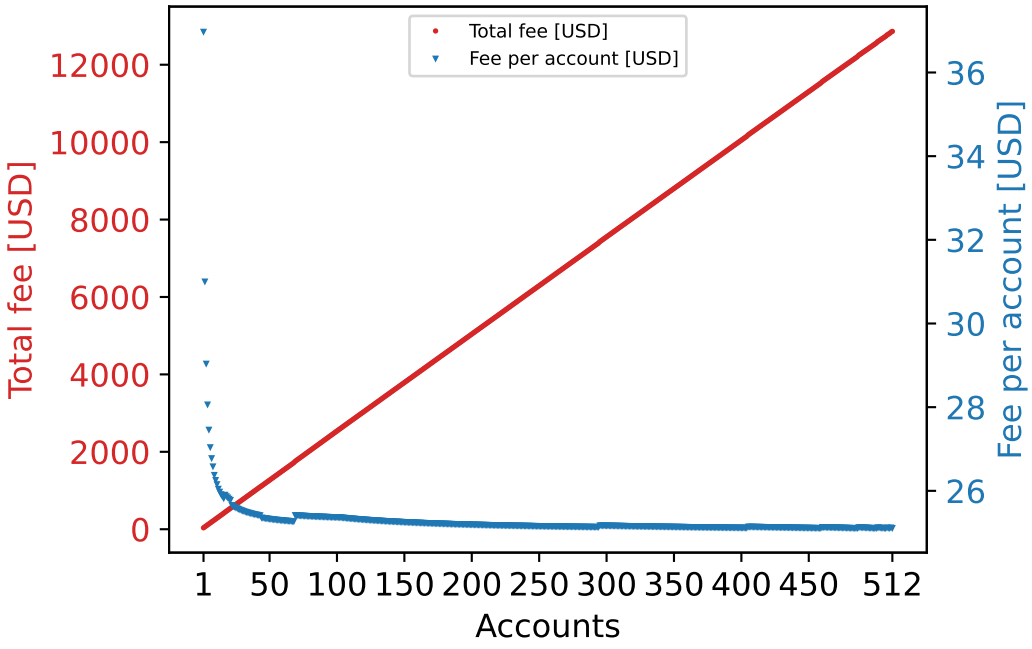

Figure 5.1: Costs of batched payout on Ethereum

Using additional storage on Ethereum results in a higher transaction fee. Therefore, when a SC's storage entry is already allocated and is updated in a transaction, the emerging fee from this transaction to allocate that storage is not charged again. Hence, whenever a developer is paid out again, the fee is smaller than in the first payout. Figure 5.2 shows the costs of batched payouts in Ethereum where all accounts have already been paid out at least once before. The costs of accounts repeatedly using the batched payout option results in an approximately 30% cheaper transaction fee than payouts for new accounts.

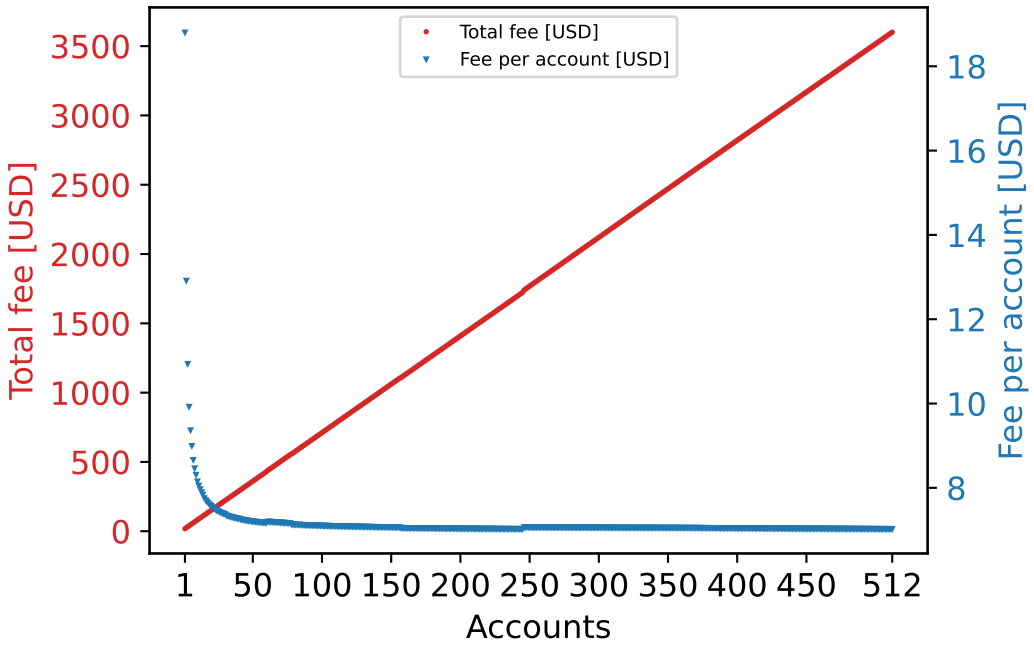

Figure 5.2: Costs of batched payout on Ethereum with existing *teas*

The cost evaluation of the `batchPayout` method on the Neo N3 SC (*cf.* Figure 5.3) presents a similar steep percental decrease in the transaction cost when including 1 to 20 developers. Only including one account in the batched payout results in a fee of USD 0.12, whereas including 20 accounts results in a fee of USD 0.089 per account. A maximum of 1012 accounts can be paid in a single transaction. This threshold is based on the maximally allowed stack items allowed in a Neo N3 transaction. Paying 1012 accounts in a transaction results in a fee of USD 0.88 per account. Thus, the decrease from including 20 accounts to 1012 accounts is only marginally smaller.

Similar to Ethereum, when paying out accounts another time after their address has been stored in the SC storage, updating that storage results in a smaller transaction fee compared to the initial payout. However, the difference is not that great, as it still costs about 90% of the initial payout costs (*cf.* Figure 5.4).

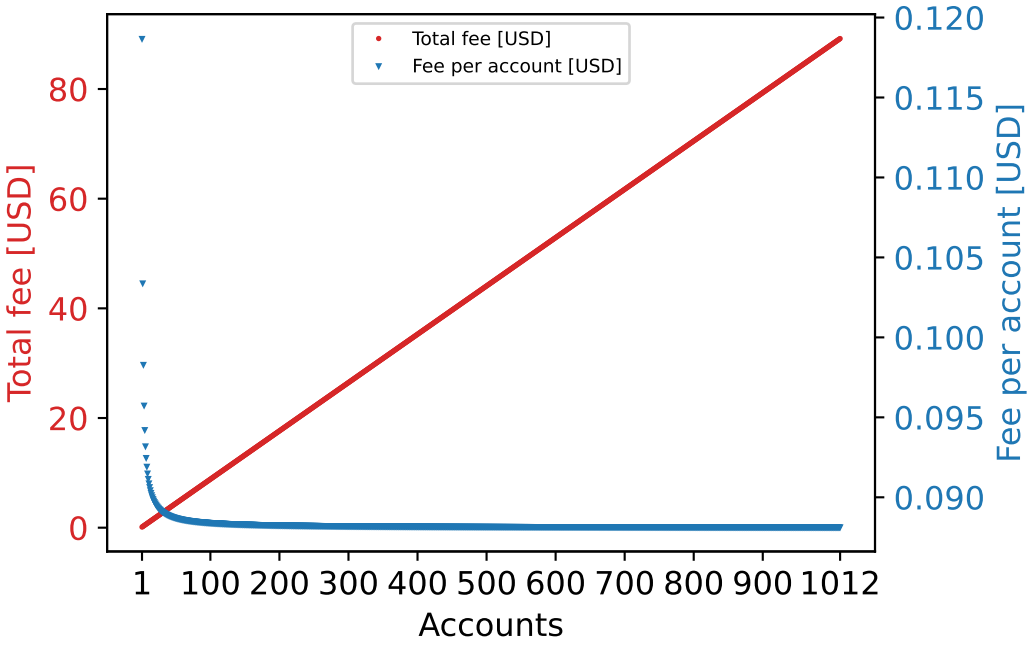

Figure 5.3: Costs of batched payout on Neo N3

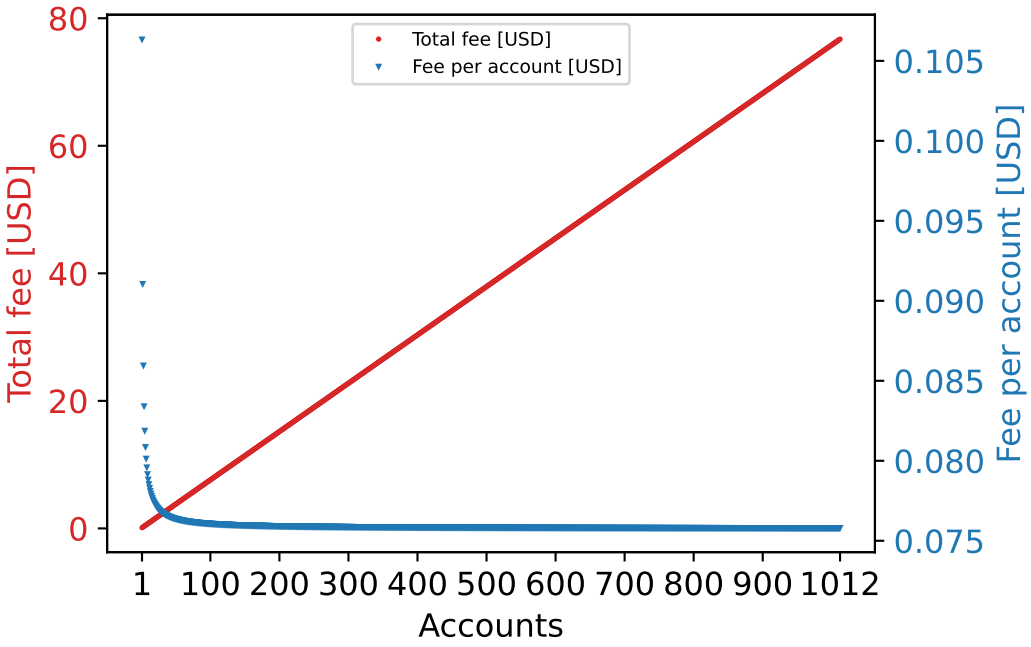

Figure 5.4: Costs of batched payout on Neo N3

As shown in Figure 5.5, the fee per account included in a batched payout in Ethereum is exceedingly higher than when using a batched payout on Neo N3. Considering including 20 accounts in a batched payout, the developers paid on Neo N3 only need to pay USD 0.089, which equals 0.34% of the fee paid when utilizing the batched payout on Ethereum (*i.e.*, USD 25.82).

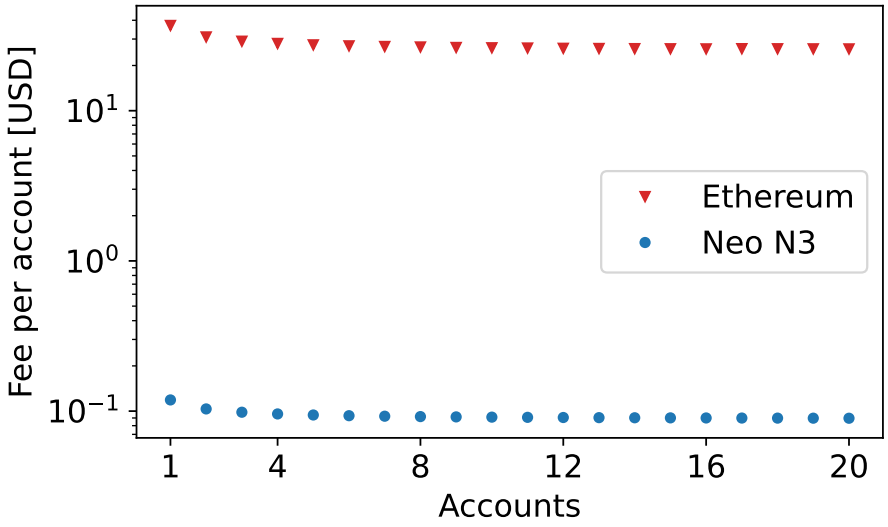

Figure 5.5: Cost of batched payout on Ethereum and Neo N3

With the signature mechanism, the proposed solution approach provides transparency without a transaction on a BC. Using a signature provided by FlatFeeStack for the first withdrawal results in the transaction costs shown in Figure 5.6. Whenever a developer withdraws again, the withdrawal cost on Ethereum is USD 20.74, while the cost on Neo N3 is USD 0.122 using a withdrawal with a signature, or USD 0.112 for a withdrawal with a pre-signed transaction, respectively (*cf.* Figure 5.7). The differences between such a transaction in Ethereum and Neo N3 are so high that the bars for a withdrawal transaction on Neo N3 are barely visible and therefore visualized in an additional subplot. The withdrawal with a signature is slightly higher due to the creation of the signed message and the call to the native *CryptoLib* SC to verify the signature while verifying the witness signature of a transaction signer is executed before the transaction script is executed, which is cheaper.

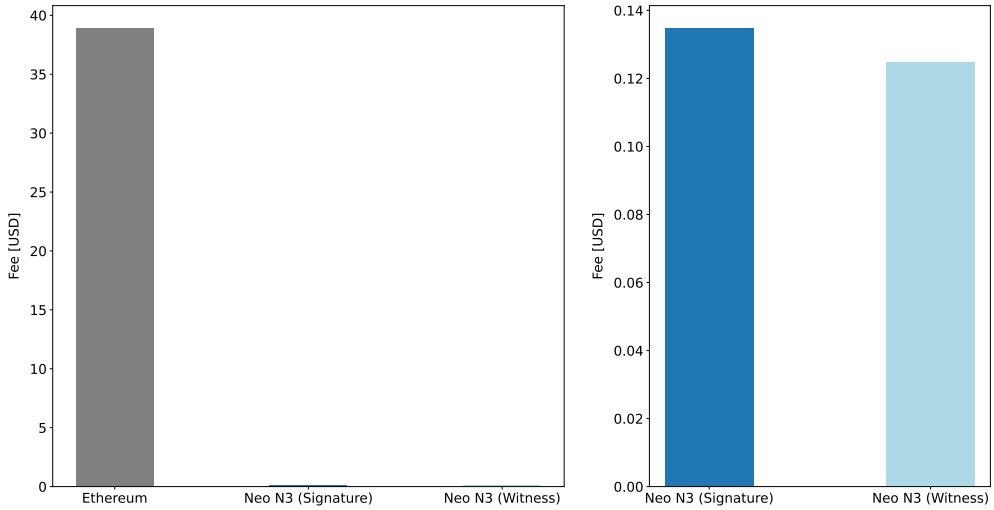

Figure 5.6: Costs of withdrawal options

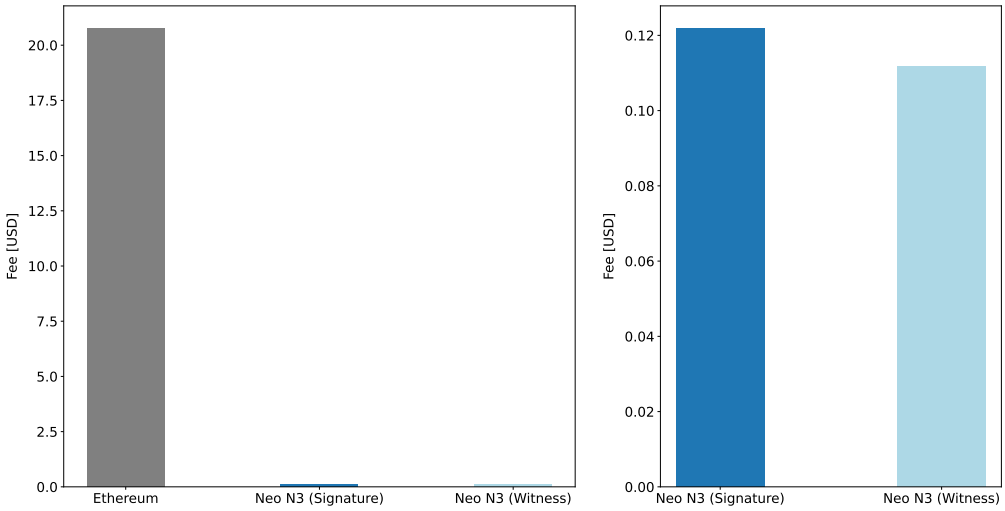

Figure 5.7: Costs of withdrawal options with existing *teas*

Figure 5.8 shows an overview of the different payout options. While the withdrawal costs the same regardless of how many developers execute a transaction, the fee per account decreases with the batched payout the more accounts are paid out in one transaction. With existing *tea* values stored on the SC, the payout is cheaper for all different payout options (*cf.* Figure 5.9).

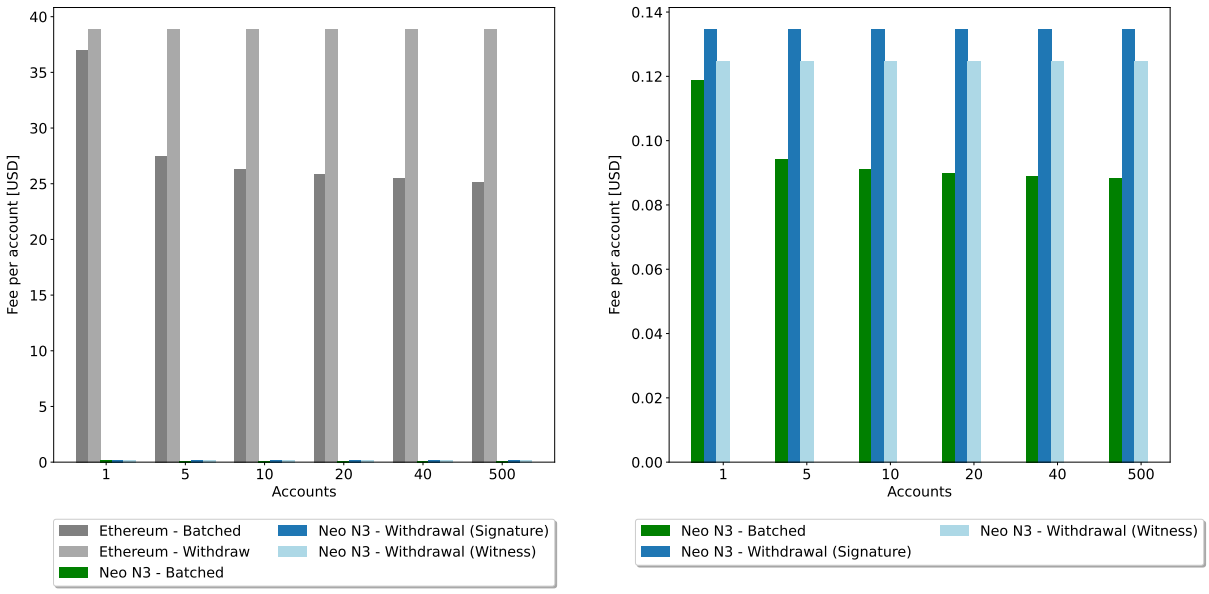

Figure 5.8: Costs of all payout options (without existing *teas*)

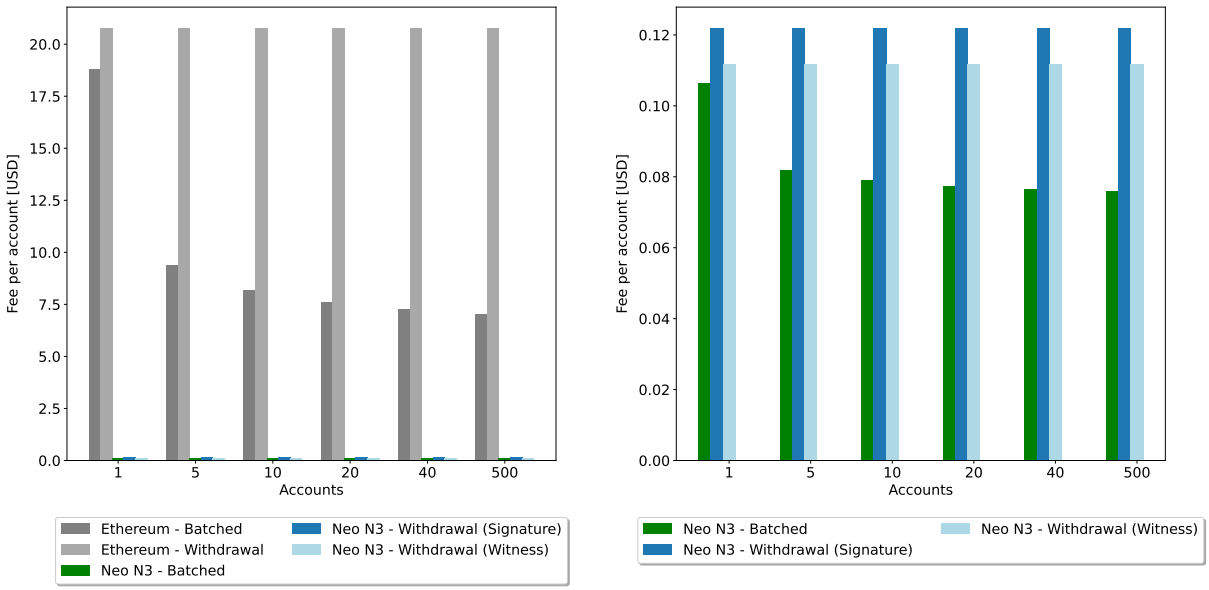

Figure 5.9: Costs of all payout options (with existing *teas*)

5.2 Initial Solution Approach

The former approach proposed by [6] is based on a *push and pull* payout mechanism as explained in Section 2.2.1. Its payout transparency consists of invoking a `fill` method on

the deployed SC that receives the token amounts and updates each developer's balance accordingly. Thus, the degree of transparency is based on how frequently an invocation of the `fill` method is executed, including this developer's information.

Figure 5.10 shows the transaction fees of invocations of the `fill` method. The decrease of transaction fees resembles the decrease of the batched payouts' fees of the proposed approach since it also consists of iterating through a list of all developer's addresses and their corresponding token amounts. Considering existing storage values the transaction fees decrease to approximately 30% of the transaction fees (*cf.* Figure 5.11).

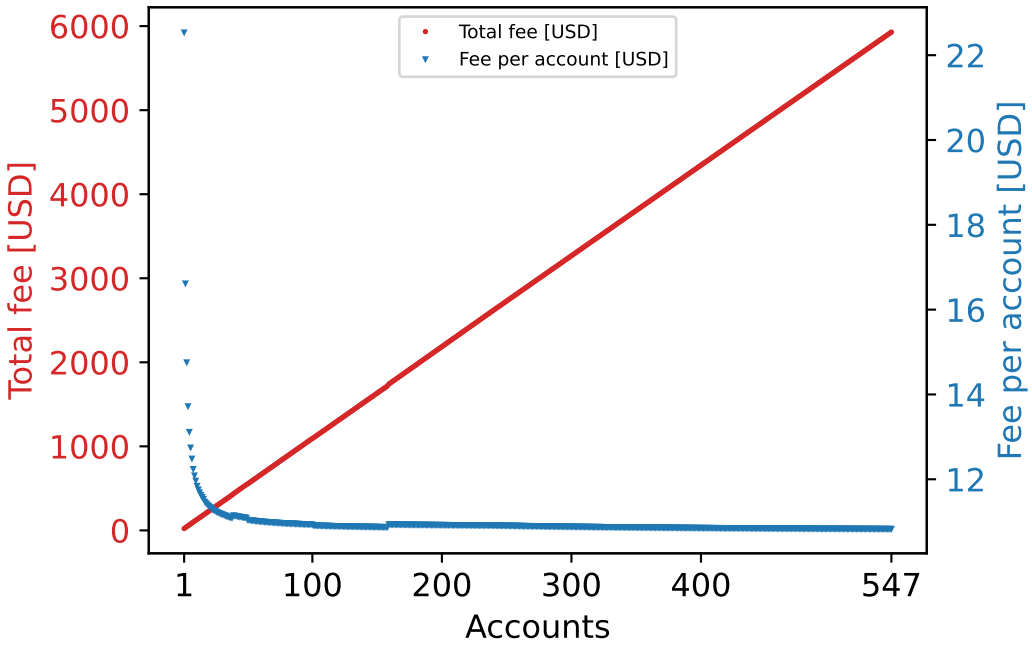

Figure 5.10: Costs of former approach's `fill` method without existing storage values

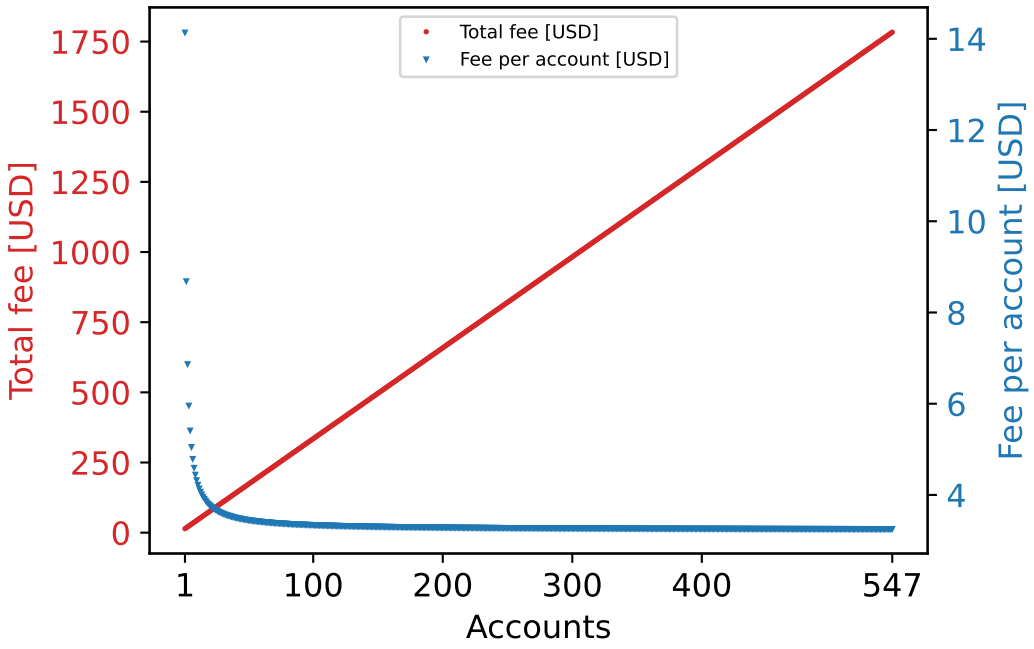

Figure 5.11: Costs of former approach's `fill` method with existing storage values

5.3 Scenario Comparison

This section shows different use case scenarios. The fees for each scenario are calculated for each payout option, *i.e.*, the former approach by [6], the proposed payout approach on Ethereum, and the proposed payout approach using Neo N3. In each scenario, it is assumed that there are a total of 100 developers. Since the payout amount does not influence the emerging transaction fees, it is assumed that the payout amount of each developer is always higher than the emerging transaction fee per account. Further, for the proposed approach on Neo N3, the withdrawal with a signature is considered since it has a slightly higher emerging fee cost. Finally, the scenarios are evaluated based on the emerging costs after one year of operation.

5.3.1 Use Case Scenario #1

In scenario 1, one hundred developers earn a token amount every week. Thus, every week the earned amounts should be communicated accordingly in a transparent way. Twenty developers wish to receive a payout every week, and eighty want to get paid every month.

The former solution approach requires FlatFeeStack to invoke the method `fill` on the former payout SC 52 times in a year to provide transparency to the developers. Twenty

developers immediately release their payment upon the updated balance of the transaction by FlatFeeStack, *i.e.*, 52 times in a year. The remaining 80 developers release their earned tokens every month, *i.e.*, 12 times a year, which results in a total of 2000 `release` invocations.

Using the proposed approach on Ethereum and Neo N3, the developers receive a signature each week, which does not incur any transaction fees. FlatFeeStack executes a batched payout every week that includes 20 developers, while in twelve of those transactions, all developers are included. Alternatively, developers could invoke the `withdraw` method to retrieve their payment, which would result in 2000 `withdraw` invocations.

Table 5.2 shows the resulting fees that emerge within one year considering the use case in scenario 1. The batched payout of the proposed approach requires only 30.34% of the transaction fees compared to the approach by [6]. Additionally, the signature mechanism provides transparency for free even though its emerging fees are still high when using a signature to withdraw earned funds.

Table 5.2: Resulting costs in scenario #1 in USD

Approach	**Blockchain**	**Payout option**	**Transparency cost**	**Total cost**
Approach by [6]	Ethereum	*Push and pull*	17'577	49'267
Thesis proposal	Ethereum	Batched payout	0	14'948
		Withdrawal	0	41'487
	Neo N3	Batched payout	0	157
		Withdrawal	0	244

The costs per developer in scenario 1 are shown in Table 5.3. For a developer that wants to be paid weekly on Ethereum, the batched payout of the proposed approach costs about 40.3%, while for monthly payouts, the fees decrease to about 23.5% compared to the former approach by [6]. Using Neo N3, the batched payout fees are significantly lowered to about 1.1% compared to the proposed approach in Ethereum. Further, the withdrawal costs only result in lower costs if used more rarely.

Table 5.3: Resulting total costs per developer in scenario #1 in USD

Approach	**Blockchain**	**Payout option**	**Weekly**	**Monthly**
Approach by [6]	Ethereum	*Push and pull*	999.72	365.91
Thesis proposal	Ethereum	Batched payout	402.76	86.17
		Withdrawal	1'078.66	248.92
	Neo N3	Batched payout	4.16	0.92
		Withdrawal	6.33	1.46

5.3.2 Use Case Scenario #2

In scenario 2, one hundred developers earn a token amount every week. Thus, every week the earned amounts should be communicated accordingly in a transparent way. Twenty developers want to move their earned tokens to their own account every week, 60 every month, and 20 every six months.

For this scenario, the former approach requires invoking the `fill` method on the former payout SC 52 times a year to provide transparency. The `release` method is invoked a total of 660 times.

Using the proposed approach, the developers receive a signature each week, and FlatFeeStack issues a batched payout transaction twice with all developers, 10 times with 80 developers and 40 times with 20 developers.

Table 5.4 shows the emerging fees within one year considering the use case in scenario 2. While the transparency of the former approach costs the same, fewer payouts result in a smaller total fee in all payout options compared to scenario 1.

Table 5.4: Resulting costs in scenario #2 in USD

Approach	**Blockchain**	**Payout option**	**Transparency cost**	**Total cost**
Approach by [6]	Ethereum	*Push and pull*	17'577	46'098
Thesis proposal	Ethereum	Batched payout	0	13'548
		Withdrawal	0	37'338
	Neo N3	Batched payout	0	142
		Withdrawal	0	219

In Table 5.5 the cost per developer in scenario 2 is shown. With the same frequency of providing transparency as in scenario 1, the developers that get paid less often in a batched payout result in a fee cost of about 6.9% compared to the former approach by [6]. Further, when using a batched payout Neo N3, the fees are reduced to about 1% of the batched payout costs on Ethereum.

Table 5.5: Resulting total costs per developer in scenario #2 in USD

Approach	**Blockchain**	**Payout option**	**Weekly**	**Monthly**	**Half-yearly**
Approach by [6]	Ethereum	*Push and pull*	999.72	365.91	207.46
Thesis proposal	Ethereum	Batched payout	403.21	86.62	14.36
		Withdrawal	1'078.67	248.92	41.49
	Neo N3	Batched payout	4.17	0.92	0.15
		Withdrawal	6.33	1.46	0.24

5.3.3 Use Case Scenario #3

In scenario 3, one hundred developers earn a token amount every day. Thus, the earned amounts should be communicated accordingly in a transparent way every day. Forty developers want a payout every month, another 40 wish to be paid quarterly, and 20 developers want to be paid at the end of the year.

For this scenario, the former approach requires invoking the `fill` method 365 times a year to provide transparency. The `release` method is invoked 660 times.

Using the proposed approach, the developers receive a signature each day. FlatFeeStack executes a batched payout transaction once with all developers, three times with 80 developers, and eight times with 40 developers. Alternatively, developers could use the `withdraw` methods instead of utilizing the batched payout. The `withdraw` method would be invoked 660 times a year.

Table 5.6 shows the emerging fees within one year considering the use case in scenario 3. With the increased demand for transparency, the former solution approach requires many additional transactions that increase its total cost to an enormous amount. The batched payout on Ethereum in the proposed approach results in a total cost of about 3.6% compared to the approach by [6]. Using the batched payout on Neo N3 further reduces the cost to 1.1% compared to the batched payout on Ethereum.

Table 5.6: Resulting costs in scenario #3 in USD

Approach	**Blockchain**	**Payout option**	**Transparency cost**	**Total cost**
Approach by [6]	Ethereum	*Push and pull*	123'379	133'836
Thesis proposal	Ethereum	Batched payout	0	4'836
		Withdrawal	0	13'690
	Neo N3	Batched payout	0	51
		Withdrawal	0	80

In Table 5.7 the costs per developer in scenario 3 are shown. The fee for developers who require the least frequent payouts results in about 0.57% of the fee cost compared to the approach by [6]. When using Neo N3, the fees are further reduced to about 1.1% compared to the proposed approach on Ethereum.

Table 5.7: Resulting total costs per developer in scenario #3 in USD

Approach	**Blockchain**	**Payout option**	**Monthly**	**Quarterly**	**Yearly**
Approach by [6]	Ethereum	*Push and pull*	1'423.93	1'297.17	1'249.63
Thesis proposal	Ethereum	Batched payout	88.46	28.85	7.18
		Withdrawal	248.92	82.97	20.74
	Neo N3	Batched payout	0.94	0.31	0.08
		Withdrawal	1.46	0.49	0.12

5.4 Discussion

The previous sections show that the proposed approach strongly indicates that on-chain fees can be reduced significantly when high transparency is desired. With the signature mechanism, transparency is no longer bound to on-chain interactions and thus reduces the overall emerging fees considerably. Further, with a batched payout as a service of FlatFeeStack, the payout fees can be lowered significantly considering that no transactions are required for transparency reasons.

Even though a withdrawal with a signature is more expensive than being included in a batched payout, signatures provide reliable protection for developers. It increases their independence and certainty of earned funds and the option to withdraw at their chosen time. Especially considering the high volatility of cryptocurrencies, the signature mechanism provides a valuable option to withdraw funds independently. Furthermore, compared to the former approach, developers are no longer dependent on the timing of a costly update of their balance. Instead, they receive full transparency for free.

Ultimately, there exist trade-offs between the batched payout and the withdrawal option. While the batched payout lowers the fee cost for the developers, they are dependent on FlatFeeStack as a centralized entity. Developers can decrease their dependence on FlatFeeStack by utilizing the withdrawal option. However, the payout cost of withdrawing is higher than in a batched payout. Nevertheless, there is no need to dispense with receiving signatures in order to increase independent certainty of earned funds from a batched payout service. Thus, combining both by utilizing a centralized triggered batched payout to optimize fee costs while still receiving signatures to remain independent, developers can benefit from the best of both worlds.

Finally, the evaluation showed that the emerging fees utilizing Ethereum are significantly higher than the emerging fees on Neo N3. Since Ethereum currently has considerably more transactions to handle and the price of ETH has increased by a factor of approximately ten within the last two years, this was to be expected. However, a potential higher demand for transactions on Neo N3 needs to be considered in the future. Nevertheless, Neo N3 provides a far more promising transactions per seconds (*tps*) output [53] than Layer-1 of Ethereum [54]. Further, in case of the GAS token price increasing significantly, Neo N3 supports multiple network factors, such as, *e.g.*, an *execution fee factor*, or a *network fee per byte* value that the Neo Council can adjust accordingly to provide reasonable transaction fees.

Chapter 6

Conclusion and Future Work

Nowadays, Open Source Software (OSS) is integrated or reused in most software projects. Nevertheless, many OSS projects are not pursued and maintained due to insufficient funding. Even though many donation platforms exist, it is cumbersome for developers to sustain an OSS project without the reassurance of financial support. FlatFeeStack is a project that aims to ease developers' exposure to intransparent funding while accommodating sponsorships to the conventional budget plan of companies.

In previous work conducted by [6], a decentralized approach by utilizing the Blockchain (BC) technology was developed in a Proof-of-Concept (PoC) implementation. It was found that the implemented *push and pull* mechanism for the payout to OSS developers is not viable due to high on-chain transaction fees. Developers that only produce small contributions would ultimately have to continue contributing to OSS without transparency of what they have earned so far.

This thesis developed a mechanism for BC-based payments to OSS contributors integrated into the FlatFeeStack project. The mechanism decreases the amount of on-chain transactions while preserving transparency to the developer. Further, it provides transparency regardless of the amount of OSS contributions by a developer. A PoC of the design has been implemented for Ethereum and Neo N3. These implementations have been thoroughly tested and optimized. The evaluation of the implementations consisted of the emerging transaction fees of the approach by [6] and the proposed payout options in real-world use cases. Further, the transaction fees on Ethereum and Neo N3 were compared for the proposed approach.

Aiming to answer the research question in Chapter 1, the findings based on the conducted evaluation and the scenarios discussed in this thesis show that it is possible to reduce on-chain transaction fees significantly compared to the former approach by [6]. Transparency is achieved by providing developers with signatures off-chain that allow them to withdraw their earned funds at any time. In order to lower the number of required transactions for a payout, a service can be provided to batch multiple payouts and thus, reduce the fee per developer to receive a payment. When high transparency is required, the proposed approach decreases the payout fees on Ethereum by 76.5% up to 99% compared to the

former approach by [6]. Under the same circumstances, utilizing the proposed approach on Neo N3 further reduces the fees by up to 99%.

In future research, the proposed design should be adopted for Layer-2 solutions on Ethereum or other Layer-1 BC systems to detect potential improvements. Furthermore, since providing signatures is highly sensitive, potential security risks (*e.g.*, Smart Contract vulnerabilities) should be assessed. Finally, surveys should be conducted to identify OSS developers' preferences toward transparency and the frequency of batched payouts.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems", 2015. Software available from tensorflow.org.

[2] OpenJS Foundation, "ESLint - Find and fix problems in your JavaScript code", 2021. `https://eslint.org/`, Last visit November 26, 2021.

[3] Rust Team, "Rust - A language empowering everyone to build reliable and efficient software.", 2021. `https://www.rust-lang.org/`, Last visit November 26, 2021.

[4] Z. Liao, B. Zhao, S. Liu, H. Jin, D. He, L. Yang, Y. Zhang, and J. Wu, "A Prediction Model of the Project Life-Span in Open Source Software Ecosystem", *Mobile Networks and Applications*, vol. 24, no. 4, pp. 1382–1391, 2019.

[5] N. Eghbal, "Roads and Bridges: History and Background of Digital Infrastructure", tech. rep., 2016.

[6] J. Brunner Zürich, "Payment Flow for an Open Source Donation Platform", tech. rep.

[7] N. Li, *Asymmetric Encryption*, pp. 142–142. Boston, MA: Springer US, 2009.

[8] P. Gallagher and A. Director, "Secure hash standard (shs)", *FIPS PUB*, vol. 180, p. 183, 1995.

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[10] Hashgard, "Smart Contract and Virtual Machine", 2019. `https://hashgard.medium.com/smart-contract-and-virtual-machine-2406edfd3dbe`, Last visit November 21, 2021.

[11] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system", *Decentralized Business Review*, p. 21260, 2008.

[12] V. Buterin, "Ethereum Whitepaper - A Next-Generation Smart Contract and Decentralized Application Platform", 2021. https://ethereum.org/en/whitepaper/, Last visit November 21, 2021.

[13] Solidity Team, "Solidity - The Solidity Contract-Oriented Programming Language", 2019. https://soliditylang.org/, Last visit November 22, 2021.

[14] Ethereum, "The history of Ethereum - London Upgrade", 2021. https://ethereum.org/en/history/#london, Last visit November 22, 2021.

[15] Buterin V., Conner E., Dudley R., Slipper M., Norden I., Bakhta A., "EIP-1559: Fee market change for ETH 1.0 chain", 2019. https://eips.ethereum.org/EIPS/eip-1559, Last visit November 22, 2021.

[16] Neo Team, "Neo - ALL IN ONE - ALL IN NEO", 2021. https://neo.org/, Last visit November 22, 2021.

[17] Neo Team, "Neo - Consensus Mechanism", 2021. https://docs.neo.org/docs/en-us/basic/consensus/dbft.html, Last visit November 22, 2021.

[18] GitHub, Inc., "GitHub Sponsors - Invest in the software that powers your world", 2021. https://github.com/sponsors, Last visit November 19, 2021.

[19] Stripe, "Stripe Connect - Payments for platforms and marketplaces", 2021. https://stripe.com/en-gb-ch/connect, Last visit November 23, 2021.

[20] GitHub, Inc., "GitHub Docs - GitHub Sponsors Additional Terms", 2021. https://docs.github.com/en/github/site-policy/github-sponsors-additional-terms, Last visit November 23, 2021.

[21] Gitcoin, "Gitcoin - Build and Fund the Open Web Together", 2021. https://gitcoin.co/, Last visit November 30, 2021.

[22] Open collective, "Open Collective - Make your community sustainable", 2021. https://opencollective.com/, Last visit November 19, 2021.

[23] Flattr AB, "Flattr - Support Creators With One Easy Subscription or With One-Time Contributions", 2021. https://flattr.com/, Last visit November 19, 2021.

[24] MANGOPAY, "MANGOPAY - Accept, secure and redistribute funds your own way", 2021. https://www.mangopay.com/, Last visit November 23, 2021.

[25] AxLabs, Axelra, coinblesk, "FlatFeeStack - On the Shoulders of Giants", 2021. https://flatfeestack.io/, Last visit November 22, 2021.

[26] C. Falter Zürich, "Design and Implementation of a Commit Evaluation Engine for an Open Source Donation Platform", tech. rep.

[27] F. Vogelsteller and V. Buterin, "EIP-20 - Token Standard", 2021. https://eips.ethereum.org/EIPS/eip-20, Last visit November 24, 2021.

[28] Etherscan, "Ethereum Average Gas Price Chart", 2021. https://etherscan.io/chart/gasprice, Last visit December 10, 2021.

[29] T. Schaffner, "Scaling public blockchains", 2021.

[30] V. Buterin, "An Incomplete Guide to Rollups", 2021. `https://vitalik.ca/general/2021/01/05/rollup.html`, Last visit December 12, 2021.

[31] Horizen, "Payment and State Channels", 2019. `https://academy.horizen.io/technology/expert/state-and-payment-channels/`, Last visit December 12, 2021.

[32] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments", 2016.

[33] Raiden Network, "The Raiden Network - Fast, cheap, scalable token transfers for Ethereum", 2021. `https://raiden.network/`, Last visit December 12, 2021.

[34] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts", *White paper*, pp. 1–47, 2017.

[35] Optimism PBC, "Optimistic Ethereum - The Ethereum you know and love, at lower cost and lightning speed", 2021. `https://www.optimism.io/`, Last visit December 12, 2021.

[36] Offchain Labs, "Arbitrum Rollup Basics", 2021. `https://developer.offchainlabs.com/docs/rollup_basics`, Last visit December 12, 2021.

[37] StarkWare, "Validity Proofs vs. Fraud Proofs", 2019. `https://medium.com/starkware/validity-proofs-vs-fraud-proofs-4ef8b4d3d87a`, Last visit December 12, 2021.

[38] Hermez, "Hermez - Scalable payments. Decentralised by design, open for everyone.", 2021. `https://hermez.io/`, Last visit December 12, 2021.

[39] StarkWare, "Starknet", 2021. `https://starkware.co/starknet/`, Last visit December 12, 2021.

[40] Neo Team, "Transaction - Structure", 2021. `https://docs.neo.org/docs/en-us/basic/concept/transaction.html`, Last visit November 28, 2021.

[41] FlatFeeStack, "`signature-provider`", 2021. `https://github.com/flatfeestack/signature-provider`, Last visit December 14, 2021.

[42] FlatFeeStack, "`payout-eth-contracts`", 2021. `https://github.com/flatfeestack/payout-eth-contracts`, Last visit December 14, 2021.

[43] FlatFeeStack, "`payout-neo-contracts`", 2021. `https://github.com/flatfeestack/payout-neo-contracts`, Last visit December 14, 2021.

[44] R. Bloemen, L. Logvinov, J. Evans, "EIP-712: Ethereum typed structured data hashing and signing", 2021. `https://eips.ethereum.org/EIPS/eip-712`, Last visit November 29, 2021.

[45] Nomic Labs LLC, "Hardhat - Ethereum development environment for professionals", 2021. `https://hardhat.org/`, Last visit November 22, 2021.

[46] E. Gesheva, "Solidity 0.6.x features: fallback and receive functions", 2020. https://blog.soliditylang.org/2020/03/26/fallback-receive-split/, Last visit November 30, 2021.

[47] S. Marx, "Stop Using Solidity's transfer() Now", 2019. https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/, Last visit November 30, 2021.

[48] AxLabs, "neow3j: A Java/Kotlin/Android Development Toolkit for the Neo Blockchain", 2021. https://neow3j.io/, Last visit November 22, 2021.

[49] AxLabs, "neo3-privatenet-docker - Run Neo N3 blockchain nodes for development in record time!", 2021. https://github.com/AxLabs/neo3-privatenet-docker, Last visit November 22, 2021.

[50] E. Zhang, "NEP-17 Token Standard", 2021. https://github.com/neo-project/proposals/blob/8b01db68bfa79ce8f23a385f7f02185ffddb8880/nep-17.mediawiki, Last visit December 1, 2021.

[51] AxLabs, "Adapt toInteger on ByteString", 2021. https://github.com/neow3j/neow3j/pull/647/files, Last visit December 15, 2021.

[52] CoinMarketCap, "CoinMarketCap", 2021. https://coinmarketcap.com/, Last visit December 10, 2021.

[53] Neo Saint Petersburg Competence Center (Neo SPCC), "Up in the mountains: reaching 50K TPS with Neo", 2021. https://neospcc.medium.com/up-in-the-mountains-reaching-50k-tps-with-neo-2f864b30abfd, Last visit December 10, 2021.

[54] Blockchair, "Ethereum - Transactions per second", 2021. https://blockchair.com/ethereum/charts/transactions-per-second, Last visit December 10, 2021.

Abbreviations

BC	Blockchain
dBFT	delegated Byzantine Fault Tolerance
DL	Distributed Ledger
EOA	Externally Owned Account
EVM	Ethereum Virtual Machine
NVM	Neo Virtual Machine
OSS	Open Source Software
PK	Private Key
PubK	Public Key
SC	Smart Contract
SDK	Software Development Kit
tea	total earned amount
tps	transactions per second
VM	Virtual Machine

List of Figures

List of Tables

Listings

Appendix A

Installation Guidelines

The code for the PoC implementations can be found on GitHub [41, 42, 43]. In the following, each repository is briefly described.

A.1 Signature Provider

The `signature-provider` repository [41] contains methods to create a signature that can be used with the implemented Ethereum and Neo N3 SCs [42, 43].

The file `signature-eth.go` contains the method `NewSignatureEth` that can be used to create a signature for the Ethereum SC. The file `signature-neo.go` provides the method `NewSignatureNeo` to create a signature that can be used for the SC on Neo N3.

Examples to create a signature with specific inputs are provided in the file `/main/main.go` which uses helper methods for Neo N3 that can be found in the file `examples-helper-neo.go`.

A.2 Ethereum Smart Contracts

The `payout-eth-contracts` repository [42] contains the PoC implementation of the Ethereum SC, tests, and helper and evaluation scripts. The SCs are located in the folder `/contracts`. The file `PayoutEth.sol` contains all basic methods while the files `PayoutEthEval.sol`, and `PayoutEthOpt.sol` inherit those methods and hold further functionalities. `PayoutEthEval.sol` has different implementation approaches that have been compared to find a fee optimized implementation. The most optimal implementation for each functionality is included in the SC written in `PayoutEthOpt.sol`.

To compile SCs and interact with a local Ethereum network for testing, the Hardhat [45] is used. The file `package.json` contains scripts that simplify the execution of Hardhat-specific commands using `yarn`.

All Ethereum SCs in the repository can be compiled with the following command:

```
$ yarn compile
```

The compiled SC information can then be found in the folder `/artifacts/contracts`.

The following command runs all tests implemented in the file `/tests/index.ts`:

```
$ yarn test
```

The repository further holds script files in the folder `/scripts` that are used as helper functions (*i.e.*, `/utils`), for manual testing (*i.e.*, `/manual-testing`), or for the evaluation (*i.e.*, `/evaluation`). The results of the evaluations can be found in the folder `/evaluation_results`.

A.3 Neo N3 Smart Contracts

The `payout-neo-contracts` repository [43] contains the PoC implementation of the Neo N3 SC, tests, helper methods, necessary code for the evaluation, and a Jupyter notebook for creating the evaluation plots. The file `PayoutNeoForEvaluation.java` has multiple different implementation approaches that have been used to find a fee optimized implementation for each required functionality. The most fee optimized implementation of each method can be found in the final SC that is represented in the file `PayoutNeo.java`.

In order to compile a SC, the `className` in the file `build.gradle` has to be changed according to the SC's class name:

```
neow3jCompiler {
    className = "io.flatfeestack.PayoutNeo"
    debug = true
}
```

The compiled artifacts can then be found in the generated folder `/build/neow3j`.

Appendix B

Contents of the CD

The attachment consists of a compressed *zip* file with the following content:

- `thesis.pdf`
- `abstract-en.txt`
- `abstract-ger.txt`
- `\thesis` (LaTeX source files)
- `\implementations`
  - `\signature-provider`
  - `\payout-eth-contracts`
  - `\payout-neo-contracts`
- `intermediary-presentation.pptx`