From Niche to Mass Adoption: Taking Native Staking Mainstream

Abstract—**Native staking is the process of using an amount of native token as a collateral in a blockchain's Proof of Stake mechanism. Ensuring accessibility to every kind of user, regardless of their wealth or knowledge, is favourable for the blockchain network and its users. However, the inherent constraints of native staking in different blockchains limit access primarily to users with substantial funds and specialist technical expertise. Additionally, existing staking pools designed to address this issue often impose fees, independent of the deposit amount, that render it infeasible to stake or unstake small amounts. In this paper, we propose a novel protocol aimed at mitigating the gas fees associated with native staking. The aim of the protocol is to combine the potential and the capabilities of both retail and whale users, in a mutually beneficial way. Moreover, we outline the design principles for another three new gas-efficient protocols and present ideas to reduce gas fees in established staking pools. As a proof of concept, we have developed a fully functional full-stack prototype for a staking pool on the Polygon network.**

Index Terms—**Web3, blockchain, native staking, proof of stake, gas efficiency, staking pools**

I. Introduction

Staking tokens involves depositing a certain native token amount as collateral within the Proof of Stake consensus mechanism of a blockchain. Proof of Stake (PoS) is a popular consensus mechanism used in blockchains to ensure the integrity of the chain and universal acceptance of a single truthful state [1].

PoS is being increasingly adopted as a consensus mechanism because it overcomes some of the significant disadvantages of traditional Proof-of-Work (PoW) consensus, such as high energy consumption [2], [3], and the requirement for large quantities of dedicated hardware which can lead to centralisation [4], [5].

PoS was initially introduced in 2012 by King and Nadal in the Whitepaper of PPCoin, also known as Peercoin [6]. They used a PoW/PoS hybrid consensus mechanism. The first blockchain to implement a pure PoS consensus mechanism was Blackcoin [7]. Following this, implementations like the PoS consensus mechanism of Algorand blockchain [4] and Ouroboros [8] for Cardano blockchain, reshaped the model of PoS mechanisms. The dynamic of PoS and its benefits over PoW influenced Ethereum, the second-most popular blockchain, to transition from a PoW mechanism to a PoS in 2022 in a move called "The Merge". This historic action achieved reducing Ethereum's energy consumption by approximately 99.5% [9].

Regardless of the changes, at its core, PoS retains the idea of having the participating nodes staking an amount, as collateral, to prevent malicious behaviour that can affect the integrity of the blockchain. Consequently, nodes that wish to participate in the PoS consensus mechanism are required to lock up an amount of native token. Then, a random process selects a node that earns the right and responsibility to add a block to the blockchain, while the rest of the network validates this new block. Every malevolent action or lazy inaction by a node, leads to a penalty which consists of losing an amount of the collateral. On the other hand, the mechanism incentivises nodes by sharing rewards for benevolent behaviour. These rewards make native staking profitable for the users and can be viewed as a low-risk source of passive income [10], [11]. At the same time, by increasing the total staked amount, the blockchain becomes more decentralised and more resistant to majority attacks since accumulating a considerable percentage of the total staked amount becomes increasingly difficult for a malicious user [12].

In theory, blockchains are meant to be decentralised and accessible to everyone following the principles set out by Nakamoto and staking should be no exception. In practice, however, staking only favours holders with substantial funds, due to high minimum deposit requirements. Thus, some service providers designed solutions based on staking pools. These combine many users' funds into a single pool mutualising transaction fees, making staking accessible to retail cryptocurrency holders [13], [14]. Ostensibly, many staking pools already exist and offer staking services to everyone without a minimum limit on the deposit amount. Upon closer examination, there are hidden gas fees that make it infeasible to stake small amounts. In fact, the gas fees are so high that if a user wishes to stake tokens with a value around 100 dollars, they would have to wait years until the accrued rewards would be enough just to cover the staking and unstaking fees.

With this motivation, we designed four different approaches for new staking pool protocols that reduce native staking's gas fees. After comparing the strengths and the drawbacks for each approach, we selected one of them and created a complete design, compatible with the Polygon network. We also developed and evaluated a full-stack prototype deployed on the Ethereum's Goerli test network as a proof of concept. Besides the design of new protocols, we believe that the core ideas proposed in this paper could be used to enhance gas efficiency in existing staking pools.

II. Preliminaries

A. Gas fees in Ethereum

Gas is a unit to measure computational labour required to execute a transaction on the Ethereum network. It is a method of payment for the node that provides the computational power needed to execute a transaction which changes the state of the blockchain. The total gas cost is calculated as the product of the amount of gas units required to execute a transaction multiplied by the gas price (i.e. cost per unit gas) [15], [16].

$$\textit{total fee} = \textit{units of gas used} \times \textit{gas price} \tag{1}$$

This paper focuses exclusively on the *units of gas used*. The *gas price* could be an interesting research topic as it depends on various factors and has significant fluctuations that affect the gas fees. However, it is beyond the scope of this paper.

The units of gas used depend solely on the computational effort required to execute a transaction. The value is invariant of the gas price and ETH's dollar value. It is also invariant to the Ethereum network on which the transaction is executed. This proves to be very useful for the unbiased comparison of the prototype that is deployed on a testnet, to the existing staking pools deployed on mainnet.

B. Pooled staking

Using a staking pool is one of the staking methods. Unlike staking as a validator node and using a staking-as-a-service platform, a staking pool usually has no minimum stake amount. Meanwhile, the provider of the staking pool relieves the users from the requirement of hardware and knowledge to run a validator node. Instead, the users delegate their funds to the pool that runs a validator node on their behalf. The main idea of pooled staking is to combine the staking power of multiple users in one node to improve the probability of being selected as validators, thus receiving more rewards. The provider receives a commission on the rewards collected by the validator node and shares the rest of the rewards to the users [17].

C. Existing staking pools

The user interacts with the staking pool by calling functions of the pool's smart contract. These calls are in the form of transactions where the user covers the gas fees. Each staking pool has its own functions but the procedures for staking and unstaking remain similar. More specifically, in order to stake, a user calls a `deposit()` function. On the contrary, a user that wishes to unstake has to call a `requestWithdraw()` function to request a withdrawal from the staked amount. After a waiting period, the funds can be claimed by the user by calling a `claimWithdraw()` function. For the Polygon network, the waiting period is 80 Polygon checkpoints, which takes approximately 2–3 days [18].

In this paper we compared the gas fees of the prototype with the gas fees of three popular staking pools. To do that we retrieved information about each pool's transactions from Etherscan and extracted the *units of gas used*. The staking pools are Lido on Polygon [17], Stader [19] and TruStake vault by Trufin [20].

D. Native staking in Polygon

There are multiple blockchains that could support the development of the proposed protocol of this paper. Polygon was selected as it has various advantages and can be used to build a prototype that could be adapted for other blockchains.

Firstly, Polygon is compatible with the Ethereum Virtual Machine (EVM) which means that it can leverage the Ethereum ecosystem standards, tools, programming languages, etc [21]. Also, Polygon's staking management contracts are deployed on Ethereum. This means that the staking procedure is made on the Ethereum mainnet and not on the Polygon mainnet.

Staking in Polygon offers an attractive Annual Percentage Yield (APY), around 5%, and has a very low minimum deposit amount, just 1 MATIC (currently around $0.77). Additionally, Polygon does not offer automatic compound rewards restaking. Users wishing to restake their rewards need to do so manually and in most cases, the cost of doing so exceeds the benefit. Instead of viewing this as a disadvantage, we view this as an opportunity to offer an automatic restaking service without requiring extra gas fees from the user.

In addition to that, unstaking in Polygon has a delay by design. Withdrawals are generally processed after an 80-checkpoint waiting period (approximately 2–3 days). Once again, this can be viewed as a chance to design a protocol that, under conditions, eliminates the waiting period.

III. Design

A. Design principles

In order to design a protocol that makes staking feasible for retail cryptocurrency holders, it is vital to reduce gas fees. As it emerges from a per-instruction *units of gas used* analysis, the high gas fees of native staking are mainly due to the interaction with the validator node. More precisely, whenever a user calls the `deposit()` function, the smart contract calls a `stake()` function that transfers and stakes the amount on the validator node. A similar procedure happens for withdrawals. Therefore, in order to mitigate the gas fees, the protocol should avoid interacting with the validator node upon every user deposit and withdrawal. It should also avoid calling the `stake()` and `unstake()` functions for small amounts as the cost of calling these functions is invariant to the amount.

Moreover, general gas-efficiency guidelines dictate that loops, complicated functions and complex data structures should be avoided. Lastly, instructions that change the state of the blockchain should be used judiciously [22].

The design approaches suggested in the following subsection have noticeable differences between them. Even so, they share some principles which are:

- There is a smart contract called a *vault* that accumulates funds from the users and interacts with the validator node by staking and unstaking. Its purpose is to act as an intermediary between the users and the validator node (see Fig. 1) and minimise the interaction with the latter. Note that funds sitting in the vault do not accrue rewards.

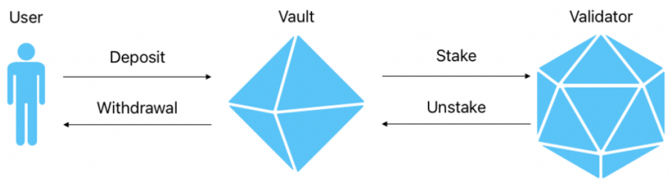

Fig. 1. Overview of the general approach.

- Users submit deposit (stake) and withdrawal (unstake) requests to the vault. From a user's point of view, depositing is staking on the validator node and withdrawing is unstaking from the validator node. In reality they are just depositing funds into and withdrawing funds from the vault. When necessary, the vault will interact with the validator node to stake or unstake.
- The vault incorporates liquid staking, whereby users deposit funds and receive *shares* represented by an ERC-20 liquidity token (LT) dedicated for the protocol. The quantity of LT held by a user remains fixed despite the accrual of staking rewards. However, the exchange rate between LT and MATIC changes.
- Requests of the opposite type, deposit and a withdrawal, can be netted. This is done by swapping a user's shares for another user's deposit amount without involving the validator node. Consequently, if a user's withdrawal request can be netted with another user's deposit request, they do not have to wait 80 checkpoints.
- Users do not pay the gas fees for calling `stake()` and `unstake()` functions on the validator node, since only the vault interacts with the validator node. Instead, they pay a small percentage fee on deposits/withdrawals. These fees are gathered and used by the vault to collectively cover the gas fees for the interaction with the validator node. Inevitably, the users also need to pay the gas fees for calling other functions of the smart contract, but these fees will be smaller.
- As Polygon does not offer automatic restaking of rewards, the protocol will use the gas fees for every call of `stake()` or `unstake()` functions, to also claim the accrued rewards. Moreover, whenever the `stake()` function is called, the claimed rewards will be restaked too.
- Users pay a fee on their rewards to the protocol, approximately 10%. This is common practice in all the other existing stakers discussed in this paper.

B. Design approaches

We consider below four candidate design approaches for the protocol and discuss the advantages and disadvantages of each one. Every approach has a different method of storing and processing the deposit and withdrawal requests but they all follow the general overview of Fig. 1.

1) Approach A. Staking Rounds:

a) Main idea: The vault has two stacks, one for deposit and one for withdrawal requests. At the end of every round, requests are netted. Fig. 2 shows an overview of the approach.

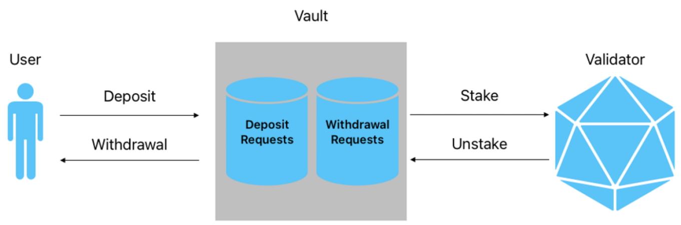

Fig. 2. Overview of Approach A.

b) Mechanics: Since the total amount in one stack will probably be larger than the other, at the end of every round, only a single validator operation needs to be performed, either stake or unstake. Additionally, in order to make the protocol self-funding, an initial deposit is made. The staking rewards accrued by that amount, during each round, will cover the fee for staking or unstaking at the end of the round.

c) Assumption: From the protocol owner's perspective, investing the initial deposit amount in the protocol needs to be more profitable than simply staking that amount. To achieve this, the profit generated from the 10% fee on the users' rewards needs to be higher than the rewards that would have accrued if the user had staked the initial deposit instead.

d) Advantages: This approach is very simple, self-funding and minimises interaction with the validator node. It also does not have a minimum staking/unstaking amount.

e) Disadvantages: The requests are not processed instantly but at the end of every round. Also, looping through the requests of each stack can be very gas-inefficient.

f) Verdict: This approach is suitable only for environments with extremely infrequent requests.

2) Approach B. S–U level:

a) Main idea: The vault has two thresholds S, U with $S > U$ and a current balance. Users interact with the vault with deposit/withdrawal requests and receive/burn shares. Fig. 3 shows an overview of the approach.

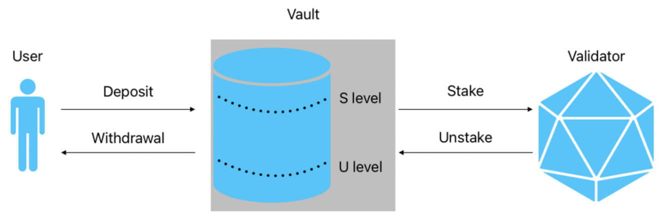

Fig. 3. Overview of Approach B.

b) Mechanics: Initially, an amount equal to S is staked on the validator and an amount equal to U is placed in the vault. When the vault's current balance reaches the threshold S, everything, except amount U, is staked on the validator node. Correspondingly, when funds in vault are insufficient to cover the withdrawal requests, an amount enough to cover the requests plus an amount U is unstaked. Meanwhile, the rewards from the staked amount in the validator are shared equally to all users regardless of whether their funds are staked in the validator or in the vault. This means that large amounts of funds sitting in the vault reduce the Annual Percentage Yield (APY).

c) Assumption: From the protocol owner's perspective, investing the initial deposit amount (equal to $S+U$) in the protocol needs to be more profitable than simply staking that amount. To achieve this, the profit generated from the 10% fee on the users' rewards needs to be higher than the rewards that would have accrued if the user had staked the initial deposit instead.

d) Advantages: This approach is self-funding and gas-efficient. It also does not have a minimum staking/unstaking amount. Moreover, the requests are processed immediately except when the current balance in the vault is insufficient.

e) Disadvantages: The design is complicated with many parameters. Additionally, a large amount of funds are constantly not staked, which lowers the average APY. Moreover, staking and unstaking whenever thresholds are reached, makes the APY non-linear, unpredictable and the protocol vulnerable to attacks.

f) Verdict: This approach is suitable for environments with balanced deposit and withdrawal requests of small amounts where the two thresholds are rarely reached.

3) Approach C. Queues with Batches:

a) Main idea: The vault has two queues with batches of requests, dQueue for deposit and wQueue for withdrawal requests. Once the total amount in a batch reaches a threshold, the batch is considered complete. If there is a complete batch of the opposite type, they are netted, else it is stored. Complete batches have an expiry date such that if they are not paired with a batch of the opposite type, they are staked/unstaked along with every other complete batch. Fig. 4 shows an overview of the approach.

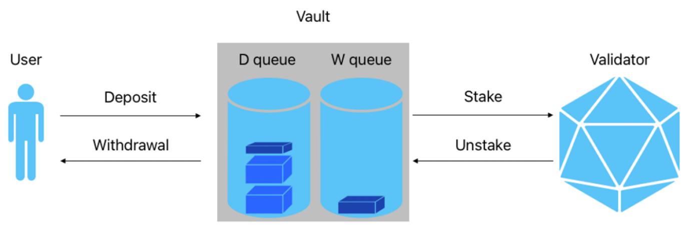

Fig. 4. Overview of Approach C.

b) Mechanics: At every point, only one of the two queues can have stored, completed batches. Also, batch size is set so that the total of the percentage deposit/withdraw fees from the users cover the cost of staking/unstaking or netting the batch.

Users get *preshares* in the vault for pending requests in the dQueue. Preshares do not accrue rewards. Once requests are processed, their preshares are swapped for shares in the validator. Meanwhile, removing a deposit request from complete batches would require rearranging the batches which would be excessively expensive. A simple solution is to prohibit a user with preshares from requesting a withdrawal.

Lastly, if there is no minimum stake/unstake amount, the number of requests in a batch can be very large. As a result, looping through the requests of the batch could be costly in term of gas fees. To keep the cost of looping through each batch low, a minimum stake/unstake amount is set. By enforcing a minimum stake/unstake amount and having a set batch size, the maximum number of requests in a batch is limited.

c) Assumption: The volume of requests exceed the batch size after a reasonable amount of time. If not, the current batches remain incomplete indefinitely.

d) Advantages: This approach is gas-efficient, self-funding and does not require an initial capital. Also, the transaction latency is low when there is a high volume of requests.

e) Disadvantages: There are two main disadvantages with this approach. The first one is that incomplete batches never expire and the second one is that users with preshares cannot withdraw. Apart from these, some transactions might be executed in parts and there is minimum stake/unstake amount. The latter, however, is not a major disadvantage as these minimum amounts are small.

f) Verdict: This approach is suitable for environments with large volumes of funds being staked and unstaked. For low volumes though, transaction latency can increase significantly.

4) Approach D. Directly Connected Queues:

a) Main idea: The vault has two queues with requests, dQueue for deposit and wQueue for withdrawal requests. Once a request is added to a queue, it is immediately netted with any pending requests on the opposite queue. When the total summed amount reaches a threshold or the oldest pending request of a queue expires before being netted, the protocol interacts with the validator node to stake/unstake. Fig. 5 shows an overview of the approach.

b) Mechanics: At every point, at least one of the queues is empty. Also, the threshold is selected so that the total of the percentage deposit/withdraw fees from the users covers the cost of staking/unstaking.

Similarly to Approach C, users get preshares in the vault for requests in the dQueue. However, if a user with preshares, requests a withdrawal, the request is netted with their pending deposit requests first (this solves the similar problem in Approach C). Furthermore, to keep the cost of looping through the requests in a queue low, a minimum stake/unstake amount

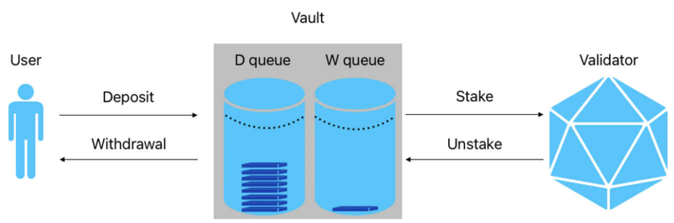

Fig. 5. Overview of Approach D.

is set. Just like in Approach C, by enforcing a minimum stake/unstake amount and having a threshold, the maximum number of requests in each queue is limited.

Finally, there are two types of deposit and withdrawal functions, the direct and the indirect ones. Retail users are likely to consider using the indirect functions while whale users are likely to consider using the direct functions.
The deposit functions are:

- `indirectDeposit()`: vault charges the user a percentage fee on their deposit amount called a deposit fee (approximately 1%). The user's request is then added to the dQueue and is processed within its expiry period.
- `directDeposit()`: the user pays the gas fees for the vault's interaction with the validator node to stake without an additional deposit fee. The user's request is processed immediately along with all the pending requests in the dQueue.

The withdrawal functions are:

- `indirectWithdraw()`: vault charges the user a percentage fee on their withdrawal amount called a withdrawal fee (approximately 1%). The user's request is then added to the wQueue and is processed within its expiry period.
- `directWithdraw()`: the user pays the gas fees for the vault's interaction with the validator node to unstake without an additional withdrawal fee. The user's request is processed immediately along with all the pending requests in the wQueue.

c) Assumption: A request expires when for a long period of time:

- there are no requests of the opposite type.
- there are no calls of the direct function of the same type.
- the total amount of the indirect requests of the same type do not exceed the threshold.

When a request expires, the vault interacts with the validator node and all the requests in the same queue are processed. Since the balance of the queue has not reached the threshold, the deposit/withdrawal fees collected from those requests are not enough to fully cover the cost of staking/unstaking. In this case, the remaining cost for staking/unstaking is paid by the protocol. It is assumed that as long as this is not common, the protocol does not require any fund injections and remains profitable.

d) Advantages: This approach is relatively simple, gas-efficient, has adjustable parameters and does not require an initial capital. It also combines the potential and capabilities of both retail and whale users in a mutually beneficial way. Furthermore, the transaction latency is generally very low. Even when it is not low, transactions are processed within a predefined time limit.

e) Disadvantages: It is not completely self-funding. In the unlikely event of continuous request expiries due to very long periods with scarce, low amounts and unbalanced requests, it might require fund injection requests. Additionally, similarly to Approach C, some transactions might be executed in parts and there is minimum indirect stake/unstake amount.

f) Verdict: This approach is suitable for environments with at least moderately-frequent requests of any type or size.

C. Final Design

Each approach has advantages and disadvantages and is more suitable for different environments. The final protocol design is based on Approach D. It appears to have the most advantages and it is also more suitable for a broad range of different environments while being relatively simple and sleek to implement.

1) Parameters: The final protocol's design relies on multiple parameters that can be modified by the owner of the protocol. The most crucial ones are shown on Table I.

TABLE I
BASIC PARAMETERS OF THE FINAL DESIGN

Parameter	Description
depositFee	percentage fee on the amount for indirect deposit
withdrawalFee	percentage fee on the amount for indirect withdrawal
minDepositAmount	minimum amount for an indirect deposit
minWithdrawalAmount	minimum amount for an indirect withdrawal
dQueueThreshold	limit of the sum of the total amounts of the requests in the dQueue for triggering the stake() function
wQueueThreshold	limit of the sum of the total amounts of the requests in the wQueue for triggering the unstake() function
expiryPeriod	time until a request expires

2) Parameter analysis:

a) Protocol's viability inequality: In order for the protocol to be viable, it is important that the funds collected from deposit fees are sufficient to cover the average staking gas fees. Therefore, the parameters $depositFee$ and $dQueueThreshold$ need to satisfy:

$$dQueueThreshold \cdot depositFee(\%) \geq \overline{stakingFee} \quad (2)$$

b) Indirect vs. direct deposit costs: The cost of indirect depositing is proportional to the deposit amount. Equation (3) shows that the total cost is the sum of the percentage deposit fee plus the `indirectDeposit()` function's inherent gas fee.

$$Cost[indirect\ deposit] = depositFee(\%) \cdot deposit + Cost[indirectDeposit()] \quad (3)$$

Direct depositing has a constant average cost for the user, invariant to the deposit amount. Equation (4) shows that the total cost is the sum of the average gas fee for staking on the validator node plus the `directDeposit()` function's inherent gas fee.

$$Cost[direct\ deposit] = \overline{stakingFee} + Cost[directDeposit()] \quad (4)$$

The cost functions (3) and (4) along with (2) can be used to create the graph shown in Fig. 6. For deposit amounts in the orange region, the cheapest option for the user is to choose the `indirectDeposit()` function which is designed for the retail users. For deposit amounts in the blue region it is cheaper to choose the `directDeposit()` function which is intended for use by whale users. $dQueueThreshold$ is selected to be equal to the threshold between the two regions. This ensures that a rational whale user should not exploit the `indirectDeposit()` to deposit a very large amount as it would be more expensive than using the `directDeposit()` function.

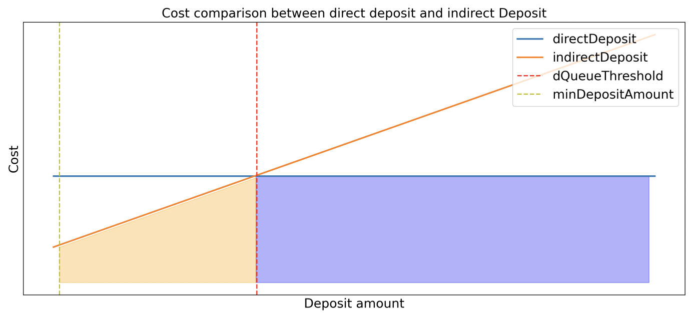

Fig. 6. Indirect vs. direct deposit costs comparison.

c) $dQueueThreshold$–$depositFee(\%)$ trade-off: As it emerges from Equation (2), $dQueueThreshold$ and $depositFee$ are inversely proportional. Ideally, users would like $depositFee$ to be as small as possible. On the other hand, if the $dQueueThreshold$ is too large, it is harder for the balance of dQueue to reach it within the expiry period, and the protocol may suffer a loss.

d) $dQueueThreshold$–$minDepositAmount$ trade-off: From the user's perspective, the minimum deposit amount for indirect deposits should be as low as possible to allow staking tiny amounts. From the protocol's perspective, low $minDepositAmount$ means large number of loops when iterating through the dQueue which increases the gas fees. The upper bound for the number of loops is given by Equation (5).

$$maxNumberOfLoops_{dQueue} = \lceil dQueueThreshold / minDepositAmount \rceil \quad (5)$$

e) $dQueueThreshold$–$expiryPeriod$ trade-off: Although users would like to have their indirect requests processed as soon as possible. Nonetheless, lowering the $expiryPeriod$ means that there is less time for the balance of dQueue to reach $dQueueThreshold$. This implies that the possibility of the collected deposit fees to be insufficient to cover the staking gas fees increases.

f) Withdrawal trade-offs: The same trade-offs that were explained for the deposits, exist for withdrawals too. The approach for selecting the values remains the same.

IV. Implementation

Based on the final design, we developed a prototype of a gas-efficient protocol. The complete repository of the prototype along with simulations for every approach and the documentation/report can be found online[1].
The basic components of the prototype are:

- **The validator node:** is the node used by protocol for staking. The vault interacts with the validator node using functions from the node's smart contract.
- **The vault:** is a smart contract deployed on the Goerli Testnet. The contract can be seen on Etherscan[2].
- **The frontend:** is a website that supports user's interaction with the vault and protocol owner's monitoring of it. The final version of the frontend is connected to the deployed smart contract and deployed using Vercel[3].
- **The autotasks:** are code scripts that automate the monitoring of the vault and initiate transactions on behalf of the protocol's owner. This prototype is monitored using four autotasks:
  - **Stake:** This autotask is triggered when the balance of dQueue reaches the $dQueueThreshold$ or when the oldest request in dQueue expires. It initiates a transaction that interacts with the validator node for staking.
  - **Unstake:** This autotask is triggered when the balance of wQueue reaches the $wQueueThreshold$ or when the oldest request in wQueue expires. It initiates a transaction that interacts with the validator node for unstaking.
  - **ExpiryCheck:** This autotask is triggered daily and checks whether the oldest requests in any queue have expired. If so, it triggers one of the previous autotasks.
  - **ClaimCheck:** This autotask is triggered daily and checks whether the waiting period of any submitted unstake request, has ended. If so, it interacts with the validator node to claim the unstaked funds and distribute them to the users.

[1]See https://anonymous.4open.science/r/Staker-E639
[2]See https://goerli.etherscan.io/address/0x9437eff6e8713cf1619d9507695489a6639b758d
[3]See https://picanha-staker.vercel.app/

V. Evaluation

A. Types of functions' executions

The developed prototype differs substantially from the existing stakers because the functions for the stake/unstake operations are not static but dynamic, thus not executed always in a similar way. For this prototype, there are not only two different types of functions for each operation (direct and indirect), but the execution cost of the same function can also vary depending on the existence of other pending requests. Figs. 7–10 show the types of execution for each of the functions that a user can call to stake or unstake. The following list analyses the gas fees for each type:

- **no pair (indirect functions):** Do not depend on any factors; gas fees for this type are relatively stable.
- **pair (indirect functions):** Depend on the number of requests of the opposite type this request is paired with.
- **stake only (direct deposit function):** Depend on the number of pending deposit requests in the dQueue
- **unstake only (direct withdraw function):** Do not depend on any factors.
- **pair only (direct functions):** Depend on the number of requests of the opposite type this request is paired with.

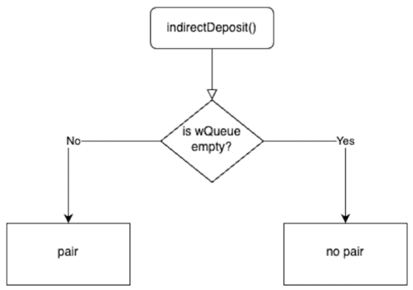

Fig. 7. Types of executions for `indirectDeposit()` function.

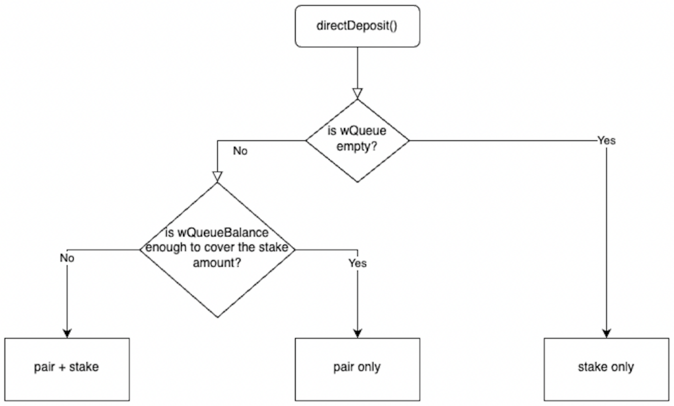

Fig. 8. Types of executions for `directDeposit()` function.

B. Experiments

For the evaluation of the prototype, we performed multiple simulations, consisted of series of transactions (txs), replicating every type of function's execution outlined in the preceding section. The average gas fee costs in gas units for every type of function execution along with the average gas fee costs for the existing staking pools can be found in Tables II and III and Figs. 11 and 12. For these data we make the assumption that a request is likely to be paired with up to 3 requests of the opposite type.

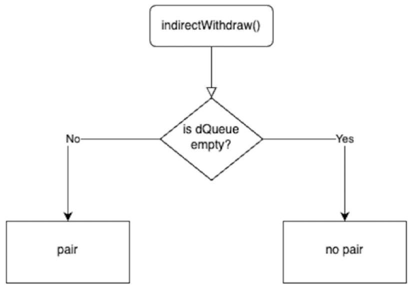

Fig. 9. Types of executions for `indirectWithdraw()` function.

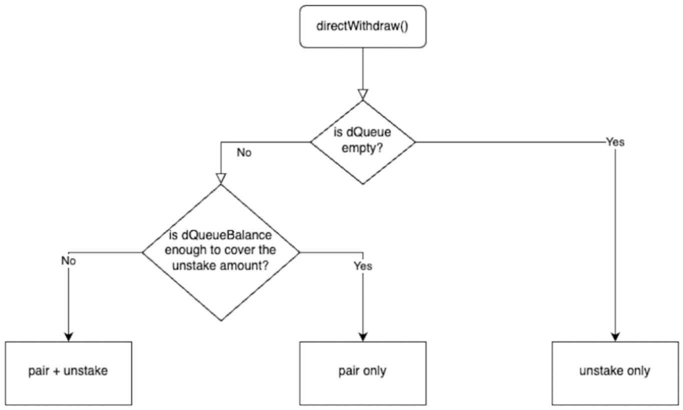

Fig. 10. Types of executions for `directWithdraw()` function.

TABLE II
Staking gas fees comparison (in gas units)

	Indirect staking		**Direct staking**		
TruStake	-		364,069		
Lido	-		642,379		
Stader	-		568,917		
	no pair	pair	stake only (X txs)	pair only (1–3 txs)	pair + stake (1 tx)
Prototype	281,518	372,717	21,615 · txs + 348,988	322,729	452,455

TABLE III
Unstaking gas fees comparison (in gas units)

	Indirect unstaking		**Direct unstaking**		
TruStake	-		609,181		
Lido	-		1,401,186		
Stader	-		947,171		
	no pair	pair	unstake only	pair only (1–3 txs)	pair + unstake (1 tx)
Prototype	268,481	369,196	559,261	335,122	601,128

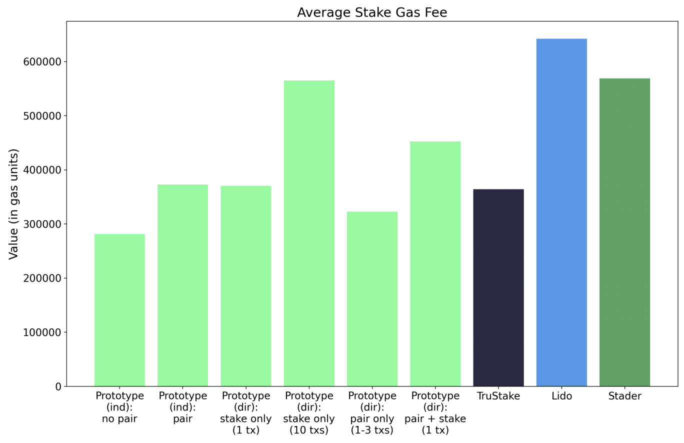

Fig. 11. Staking gas fees comparison.

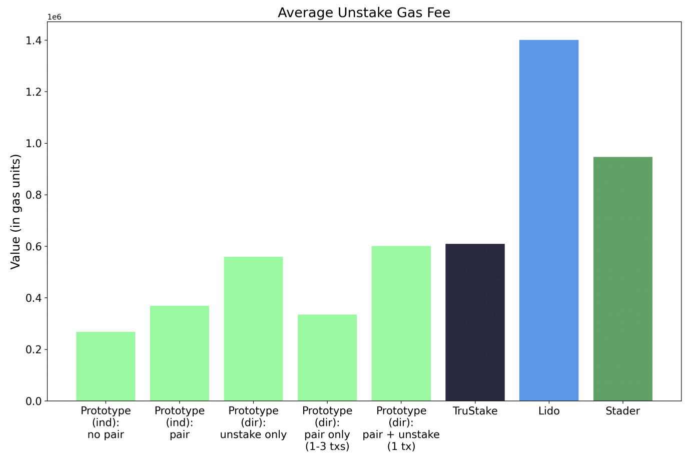

Fig. 12. Unstaking gas fees comparison.

C. Findings

The prototype achieves its primary goal of reducing the gas fees compared to the existing staking pools. Fig. 12 shows that for unstaking, the prototype is always cheaper than any other staker. Fig. 11 shows that for staking, it is generally, but not always, cheaper than the other stakers.

Meanwhile, the protocol proves that although gas fees can be reduced, they cannot be eliminated. Even by avoiding frequent interaction with the validator node, the user is required to cover the gas fees of a function that interacts with the vault.

A way to reduce the staking gas fees is to split the staking and the share minting procedures into different functions. This will make the `directDeposit()` function invariant to the number of requests in the dQueue for stake only. We suggest that the concept of preshares is dropped completely and every user immediately receives freshly minted shares for every direct or indirect deposit. It is reasonable to assume that the total staked amount will be significantly larger than the $dQueueThreshold$; therefore, the APY will not be affected significantly by the removal of the preshares concept. This will also solve the withdrawal problem of Approach C and simplify the smart contract's function thus further reduce the gas fees.

Additionally, we concluded that the values of the parameters of Table I should be reevaluated frequently to adjust to the changes of the gas and cryptocurrency prices and the volume of vault transactions.

D. Discussion

We believe that the protocol we propose promotes the benefits of native staking and supports the needs of every kind of user in an inclusive and mutually beneficial way.

For retail users, it provides access to native staking in a simple, quick and cheap way. For whale users, it provides low fees, complimentary fund claiming and, is some cases, instant unstaking bypassing the waiting period. More importantly, it does not discriminate against any class of users as everyone has the same rights and advantages.

Moreover, the proposed protocol can be more profitable for its owner than than the existing stakers. While the other stakers only rely on the 10% fee on the rewards, the proposed protocol can generate an additional profit. When indirect requests are netted, their percentage fees are not spent to interact with the validator node but kept as profit for the protocol.

Lastly, it promotes native staking making the blockchain more decentralised and resistant to malicious attacks.

VI. Future work

The suggestions for future work can be split into two categories: improvements on the proposed prototype and use of this work as a stepping stone for further research on gas-efficiency and native staking.

Concerning the former, we believe that the prototype can be improved by analysing the behaviour of the gas price. Furthermore, the protocol can be redesigned to remove the concept of preshares and reevaluate the parameter values frequently as explained in Section V-C. Lastly, limiting the maximum number of pairings for a request can avoid rare situations that can potentially increase the gas fees.

As for the latter, we recommend that our work can be used for the design and development of gas-efficient protocols for native staking on other blockchains. Additionally, we suggest that the approaches discussed in this paper can be combined to create a hybrid protocol that adjusts to the environment i.e. the types and the volume of the requests. Finally, we believe that some of the ideas of this paper can be integrated into existing staking pools to reduce their gas fees.

VII. Conclusion

Through this paper we illustrated that, even though completely eliminating native staking gas fees is infeasible, it is possible to mitigate the gas fees and make native staking genuinely accessible to all users. We suggested several gas-efficient approaches and design principles that can be used for new or existing native staking pools. Finally, we developed and evaluated a fully functional full-stack prototype that achieved lower gas fees than the state-of-the-art staking pools handling assets worth hundreds of millions of dollars.

References

[1] Ethereum, "Consensus mechanisms," [Accessed:18/05/2023]. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/

[2] F. Saleh, "Blockchain without waste: Proof-of-Stake," *The Review of Financial Studies*, vol. 34, pp. 1156–1190, 3 2021. [Online]. Available: https://doi.org/10.1093/rfs/hhaa075

[3] B. Biais, C. Bisiere, M. Bouvard, and C. Casamatta, "The blockchain folk theorem," *The Review of Financial Studies*, vol. 32, pp. 1662–1715, 2019.

[4] J. Chen and S. Micali, "Algorand," 7 2016. [Online]. Available: http://arxiv.org/abs/1607.01341

[5] Ethereum, "Proof-of-work (PoW)," [Accessed:20/05/2023]. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/

[6] S. King and S. Nadal, "PPCoin: Peer-to-peer crypto-currency with proof-of-stake," 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:42319203

[7] P. Vasin and B. Co, "BlackCoin's Proof-of-Stake Protocol v2," 2014. [Online]. Available: www.blackcoin.co

[8] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol." Springer, 2017, pp. 357–388.

[9] Ethereum, "The merge," [Accessed:22/05/2023]. [Online]. Available: https://ethereum.org/en/roadmap/merge/

[10] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," 3 2020. [Online]. Available: http://arxiv.org/abs/2003.03052

[11] Ethereum, "Proof-of-stake (PoS)," [Accessed:18/05/2023]. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[12] ——, "Ethereum staking," [Accessed:17/05/2023]. [Online]. Available: https://ethereum.org/en/staking/

[13] L. Brünjes, A. Kiayias, E. Koutsoupias, and A.-P. Stouka, "Reward sharing schemes for stake pools," *CoRR*, vol. abs/1807.11218, 2018. [Online]. Available: http://arxiv.org/abs/1807.11218

[14] H. Gersbach, A. Mamageishvili, and M. Schneider, "Staking pools on blockchains," 3 2022. [Online]. Available: http://arxiv.org/abs/2203.05838

[15] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform." 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[16] Ethereum, "Gas and fees," [Accessed:11/08/2023]. [Online]. Available: https://ethereum.org/en/developers/docs/gas/

[17] Lido, "Overview — Lido On Polygon Docs," [Accessed:30/05/2023]. [Online]. Available: https://docs.polygon.lido.fi/

[18] Polygon, "MATIC Staking — Stake MATIC tokens for network rewards," [Accessed:01/08/2023]. [Online]. Available: https://polygon.technology/staking

[19] StaderLabs, "Polygon — StaderLabs docs," [Accessed:30/05/2023]. [Online]. Available: https://www.staderlabs.com/docs-v1/category/polygon

[20] TruFin, "TruStake (MATIC) Staker - TruFin Documentation," [Accessed:29/05/2023]. [Online]. Available: https://trufin.gitbook.io/docs/trustake-vaults/trustake-matic-staker

[21] Polygon, "Polygon lightpaper," 2021. [Online]. Available: https://whitepaper.io/document/646/polygon-whitepaper

[22] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020, pp. 9–15.