An Efficient Micropayment Channel on Ethereum

Hisham S. Galal, Muhammad ElSheikh, and Amr M. Youssef

Concordia Institute for Information Systems Engineering,
Concordia University, Montréal, Québec, Canada

Abstract Blockchain protocols for cryptocurrencies offer secure payment transactions, yet their throughput pales in comparison to centralized payment systems such as VISA. Moreover, transactions incur fees that relatively hinder the adoption of cryptocurrencies for simple daily payments. Micropayment channels are second layer protocols that allow efficient and nearly unlimited number of payments between parties at the cost of only two transactions, one to initiate it and the other one to close it. Typically, the de-facto approach for micropayment channels on Ethereum is to utilize digital signatures which incur a constant gas cost but still relatively high due to expensive elliptic curve operations. Recently, ElSheikh *et. al* have proposed a protocol that utilizes hash chain which scales linearly with the channel capacity and has a lower cost compared to the digital signature based channel up to a capacity of 1000 micropayments. In this paper, we improve even more and propose a protocol that scales logarithmically with the channel capacity. Furthermore, by utilizing a variant of Merkle tree, our protocol does not require the payer to lock the entire balance at the channel creation which is an intrinsic limitation with the current alternatives. To assess the efficiency of our protocol, we carried out a number of experiments, and the results prove a positive efficiency and an overall low cost. Finally, we release the source code for prototype on Github[1].

Keywords: Micropayment Channel, Ethereum, Merkle Tree.

1 Introduction

Cryptocurrencies such as Bitcoin [8] and Etheruem [11] enable secure payment transactions between parties using blockchain technology. The major innovation that made the success of Bitcoin was the Nakamoto consensus [8] that utilizes Proof of Work (PoW) to enable distrusting peers to reach consensus. However, this level of security comes at the cost of limited throughput and delayed confirmation. For example, the transaction throughput in Bitcoin and Ethereum are roughly 5 and 20 transactions per second [1], respectively. Furthermore, it was shown [5] that blockchain protocols based on PoW can hardly scale beyond 60 transactions per second without considerably weakening their security.

[1] `https://github.com/HSG88/Payment-Channel`

Additionally, the transaction fees are not constant and they vary significantly based on the current price of the underlying coin, and also whether the network faces high traffic of transactions. These limitations make applications such as micropayments expensive to realize directly without further modification to the consensus protocol.

A payment channel [3,6,7,9] is a protocol between two parties to send nearly an unlimited number of payments interactions off-chain. To establish it, only two transactions are required, one to open the channel (i.e., creating a smart contract and funding it), and the second one to close it (i.e., reclaiming the funds). Furthermore, a channel can have certain properties such as being a unidirectional and monotonically increasing which allows us to have entirely off-line channel. In other words, none of parties have to stay on-line to monitor changes on the smart contract. All they have to watch for is the channel timeout. Furthermore, a unidirectional and monotonically increasing payment channel is convenient for a merchant and buyer scenario.

The de-facto standard payment channel protocols on Ethereum depend heavily on digital signature schemes which incur a constant cost, yet a relatively high one due to the cost of elliptic curve operations to verify the signature. Recently, ElSheikh *et. al* [4] have proposed `EthWord`, a protocol that utilizes hash chain to create an efficient payment channel as it scales linearly with the channel capacity (i.e., number of unit payments), and it has a lower gas cost up to roughly a channel capacity of 1000 units of payments compared to the digital signature based channels.

In this work, we improve even more on the efficiency and gas cost of the above protocols and propose a scheme which scales logarithmically with the channel capacity. Furthermore, the payer does not have to lock up the entire amount in the construction of the payment channel, which is an intrinsic limitation in the `EthWord` and digital signature based channels. Furthermore, as part of our contribution, we also provide an open-source prototype on Github (`https://github.com/HSG88/Payment-Channel`) for the community to review it.

The rest of this paper is organized as follows. Section 2 provides a brief review of a digital signature based channel construction referred to as `Pay50`, and the hash chain based channel `EthWord`. In Section 3, we present the cryptographic primitives utilized in the proposed scheme. Then, in Section 4, we provide the design for our approach and compare its efficiency and cost to the other constructions. Finally, we present our conclusions in Section 5.

2 Related Work

Pay50. To argue the simplicity of building payment channels on Ethereum, Di Ferrante [2] showed how to construct one using only '50 lines of code' Solidity implementation of a uni-directional, monotonic channel. Simply, the payer initializes a smart contract with the payee's address, a timeout value, and deposits some balance. Once it is deployed, the smart contract constructor stores the payer's address in order to verify the digital signatures of off-chain payments via

calling `ecrecover` (an op-code in Ethereum that returns the signer's address). After the deployment, the payer can send digitally signed payments messages to the payee off-chain. The payee verifies the digital signature, and on success, the payee provides the service or the item to the payer. At a later point of time but before the channel timeout, the payee sends the last signed payment message to the smart contract which releases the amount on successful verification.

EthWord [4] builds mainly on PayWord [10] and it depends significantly on a hash chain which is constructed by iteratively applying a public one-way hash function H on a secret random number s. More precisely, assume that $H_i(s) = H(H_{i-1}(s))$ for $i \in [1, n]$ (i.e, the length of the hash chain is n). The last hash value in the chain $H_n(s)$ is referred to as the *tip*. Furthermore, since the utilized hash function H is assumed to have the preimage resistance property, then it is computationally infeasible for an adversary to find any preimage in the chain given its *tip*. In the context of payments, the payer creates a hash chain of length n which represents the maximum number of possible payments. Then, the payer creates a channel by deploying a smart contract on Ethereum, passing the *tip*, the payee address, and timeout value as parameters. Later, the payer can send units of payments by revealing a preimage value deeper in the chain. For example, to pay i units, the payer sends $H_{n-i}(s)$ to the payee. The payee can verify it off-chain by iteratively hashing $H_{n-i}(s)$ i times, and see if the result equals the *tip*. Before the channel timeout, the payee sends the latest H_{n-i} to reclaim i units once the smart contract has verified it. Similar to `Pay50`, after the timeout, the smart contract sends the remaining balance back to the payer.

3 Preliminaries

Merkle Tree. is a core component of the blockchain protocols. In Ethereum [11], Merkle trees aggregate the hashes of transactions, states, and receipts in a particular block so that the root becomes a binding commitment to all these values in that block. Technically speaking, a Merkle tree is a balanced binary tree in which the leaf nodes hold values and each non-leaf node stores $H(LeftChild||RightChild)$, where H is a collision-resistant hash function. Proving the membership of a value in the tree can be achieved with a logarithmic size proof (in terms of the number of leaves), known as a *Merkle proof*. For example, given a Merkle tree M with a root r, to prove that a value $x \in M$, the prover creates a Merkle proof π by retrieving the siblings of each node on the path from x to r. The verifier iterates over the nodes in π_i to construct a root r' and accepts the proof if $r' = r$. It is worth noting that since M is a binary tree, then the proof size $|\pi| = log_2(n)$ as shown in Fig. 1.

The extension to our scheme that enables the payer to add funds to the channel depends on a variant of Merkle tree which is not strictly balanced. The objective is to append new values to the tree while at the same time maintain correct Merkle proofs for the old values. More precisely, to append a new set of m values to an existing Merkle tree M_n with n leaves and root r_n. First, we generate a Merkle tree M_m with a root r_m from the m values. Then, we combine

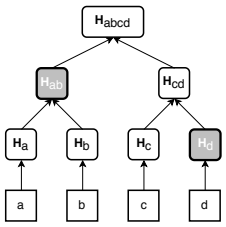

Figure 1: An example illustrating the Merkle proof for element $c \in M$ which consists of the nodes H_d and H_{ab}

the Merkle trees M_n and M_m to generate a new tree M' that contains the roots r_n and r_m as the child nodes of its root r'. Note that we still preserve the correctness of Merkle proofs for values in M_n by augmenting any valid Merkle proof π for elements in M_n with the root r_m to be valid with respect to M'.

4 Protocol Design and Implementation

Assume that Alice runs an online service for which she accepts (ether) currency on a micro-level (e.g., a very low fraction of `ether` that costs less than one dollar). Bob is interested in that service and he wants to utilize it. However, sending transactions to the Ethereum blockchain has a minimum cost of 21000 gas [11] which becomes too expensive for Bob as the number of interactions between him and Alice increases. To make it efficient and also cheap, Alice and Bob can utilize an off-chain payment channel. There are three phases in our protocol from the initiation to completion.

4.1 Channel Setup

This phase starts with Bob generating a secret random number s_0. Suppose that he estimates the number of maximum units of payments (i.e., channel capacity) he is willing to make is n. Note that we explain later how he can increase this number in case he wants to utilize the service more than he expected. Then, Bob uses a pseudo random number generator with the seed s_0 to create a sequence of random numbers $(s_1, ..., s_n)$). After that, he creates a Merkle tree M to bind the elements $((1||s_1), ..., (i||s_i), ..., (n||s_n))$ where every element is the concatenation of a value $i \in [1, n]$ and a random number s_i. We also assume that there is a minimum unit of payment u, (e.g., $u = \texttt{GWei} = 1 \times 10^{-9}$ `ether`).

At this moment, Bob deploys a smart contract on Ethereum to act as a trusted third party that holds Bob's balance and settles the final payment transaction to Alice. To deploy it, Bob passes the following parameters that control the payment channel between him and Alice to the smart contract constructor:

1. Alice's address A_{adr} on Ethereum.

2. A timeout value T_{out} before the channel is closed.
3. The root r of the Merkle tree M.

Additionally, Bob has to deposit an amount $balance = n \times u$ `ether` in the smart contract to be held in escrow, and pay Alice when she submits a valid Merkle proof.

4.2 Off-chain Payments

Every time Bob wants to utilize Alice's service, he sends her a new Merkle proof π_i for an amount of i units. To generate the proof π_i, Bob runs Algorithm 1 which takes a Merkle tree M, an amount i, and the seed value s_0 as parameters. We

Algorithm 1 Create Merkle Proof π_i for an amount i and random seed s_0

```
1: function CreateMerkleProof(M, i, s_0)
2:     π_i ← []
3:     s_i ← PRNG(s_0, i)
4:     node = H(i||s_i)
5:     while node ≠ M.root do
6:         neighbour ← M.GetNeighbour(node)
7:         π_i.Append(neighbour)
8:         node ← M.GetParent(node)
9:     end while
10:    return π_i
11: end function
```

start by generating the corresponding random value s_i. Then, s_i is concatenated to the amount i before feeding the result to the hash function `H`. Doing so prevents Alice from guessing the pre-images of the leaves in MT. In our implementation, we utilize `Keccak256` as the hash function `H` due to its built-in support in the Ethereum virtual machine as an op-code. Then, we append the neighbour node of each node in the path from the leaf to the root r to the proof π_i.

After receiving the proof π_i, Alice has to verify it before providing the new service to Bob. So she calls Algorithm 2, and she decides to accept or reject based on the output. Note that, Algorithm 2 is also one of the functions in the smart contract that settles the payment to Alice. Essentially, Alice has to only keep track of the latest π_i (i.e., the proof for largest i amount). Once, Alice and Bob agree that there is no more payment interactions going between them and before the channel timeout, Alice sends π_i to the smart contract which will verify it and releases the payment to Alice.

4.3 Channel Termination

At this phase, Alice cannot issue any payment request to the smart contract as the channel has reached its timeout. However, Bob can reclaim his remaining

Algorithm 2 Verify Merkle Proof π_i for a Merkle tree with root r, amount i, and a random number s_i

```
1: function VerifyMerkleProof(π_i, i, s_i, r)
2:     node = H(i, s_i)
3:     for j ← 1 to size(π_i) do
4:         if node < π_{i,j} then
5:             node ← H(node||π_{i,j})
6:         else
7:             node ← H(π_{i,j}||node)
8:         end if
9:     end for
10:    if node = r then
11:        return true
12:    else
13:        return false
14:    end if
15: end function
```

funds by calling `selfdestruct`. Typically, this will disable the smart contract from processing any further transactions. However, the smart contract can be alternatively designed to allow for reusing it for new payment channels without destructing it. Nonetheless, in our implementation, we chose a similar design to `Pay50` and `EthWord` in order to make fair comparisons.

4.4 Dynamic Refund Extension

One major advantage for our approach compared to the other alternatives is the ability to add more fund as Bob wants while the channel is open. In other words, Bob is not required to lock the full amount of balance which can be a substantial value at the time the channel construction. Toward this end, when required, Bob creates a new Merkle tree M_2 with a root r_2 that binds the additional m amounts of payments. Then, he sends r_2 to the smart contract along with the additional balance. The smart contract combines the old root r_1 with r_2 and hash the result to create a new root r'. Note that, since Ethereum is a public blockchain, then Alice can see the transaction carrying r_2. Therefore, she can still generate a valid proof for previous payments by augmenting the r_2 to the Merkle proof π_i for $i \in [1, n]$. In other words, Alice can still guarantee that she can reclaim her latest payments even if Bob acted maliciously and generated a bogus root r_2 for a fake tree M_2.

4.5 Evaluation

To assess our approach, we created experiments to check the efficiency and gas cost associated with `Pay50`, `EthWord`, and our approach. We also created off-chain clients to interact with the smart contract of each one. To our expectation, the

efficiency of our approach as indicated by its gas cost far outweighs the corresponding cost of `Pay50` and `EthWord`. The results shown in Fig. 2 indicate that

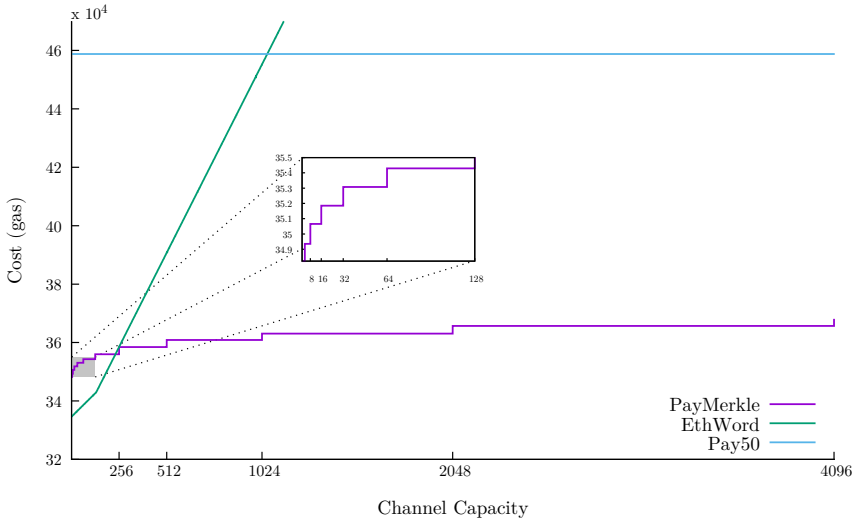

Figure 2: The total gas cost of payment channel creation and settlement for different channel capacities.

our approach is better than `Pay50` virtually on all practical channel capacities, however, it is slightly behind `EthWord` when the the channel capacity is lower than 256, due to the added cost of Merkle proof size and the code logic to verify it. However, our approach scales much better after that level as the gas cost of `EthWord` increases at a much faster pace, and stays behind `Pay50` once the channel capacity is above 1000. Interestingly, invoking the dynamic refund extension in our solution costs 28,941 gas which is much cheaper than the cost to deploy a new `EthWord` payment channel (318,953 gas).

It is also worth noting that the size of the Merkle proof is taken into account by the gas cost of the channel termination transaction. Hence, by achieving an overall lower gas cost, this implies that the cost for the increased parameters size in our protocol compared to the constant single parameter in `EthWord` and `Pay50` is paid off by the efficiency of verifying the Merkle proofs compared to processing the hash chain in `EthWord` and verifying digital signature in `Pay50`. Concretely, the largest non-realistic capacity is 2^{256}, which requires a Merkle proof of 256 hashes. Therefore, the Merkle proof size in bytes is 256×32 bytes $=$ 8-Kbytes. From the yellow paper of Ethereum, the fee for every non-zero byte is 68 gas. As a result, the maximum possible Merkle proof size will incur a total gas cost of $557,056$ gas which is approximately 8% of the current block gas limit.

5 Conclusion

One way to improve the throughput of PoW based blockchains is by utilizing second layer improvements such as payment channels. Furthermore, payment channels increase the adoption of cryptocurrencies for simple payments as it requires only two transactions to be committed to the blockchain while allowing an unlimited number of transactions off-chain. The de-facto standard for payment channels is to utilize digital signatures which incur a constant cost, yet relatively high one compared to hash-based solutions such as `EthWord`. In this work, we further improved upon the efficiency of these schemes and proposed a solution, based on Merkle trees, that scales logarithmically with the number of payments. Additionally, our approach does not require the payer to lock the entire amount of balance at the time of the channel construction. In other words, the payer can add up funds at later points of time at a much cheaper cost than recreating new payment channels as found in previous proposals such as `Pay50` and `EthWord`. For future work, we will investigate how to scale our approach to build payment networks between multiple parties, and also add the ability to make a bidirectional channel.

References

1. Blockchain explorer: Number of transactions per day in bitcoin and ethereum, 2019. `https://www.blockchain.com/explorer`.
2. M. Di Ferrante. Ethereum payment channel in 50 lines of code. Medium, June 2017.
3. S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 327–344, 2019.
4. M. Elsheikh, J. Clark, and A. M. Youssef. Deploying payword on ethereum. In *International Conference on Financial Cryptography and Data Security Workshops, BITCOIN, VOTING, and WTSC (To Appear)*. Springer, 2019.
5. A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16. ACM, 2016.
6. L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. Sok: Off the chain transactions. *IACR Cryptology ePrint Archive*, 2019:360, 2019.
7. A. Miller, I. Bentov, R. Kumaresan, C. Cordi, and P. McCorry. Sprites and state channels: Payment networks that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.
8. S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.
9. J. Poon and T. Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.
10. R. L. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *International workshop on security protocols*, pages 69–87. Springer, 1996.
11. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.