Payment Channels Under Network Congestion

Tuan Tran
UC Santa Cruz
atran18@ucsc.edu

Haofan Zheng
UC Santa Cruz
hzheng6@ucsc.edu

Peter Alvaro
UC Santa Cruz
palvaro@ucsc.edu

Owen Arden
UC Santa Cruz
owen@soe.ucsc.edu

***Abstract*—Sending transactions on leading blockchains such as Ethereum can be slow and costly. A payment channel is a well-known scaling solution that minimizes transactions sent on the chain, and allows users to transact more efficiently. One of the guarantees of payment channels is that there is no counterparty risk, so an honest party is able to withdraw the amount of money that is reflected by the most recent transaction agreed by both parties. In this paper, we show that this guarantee can be violated when the network is under congestion. Regardless of whether or not the honest party is online, the malicious party can leverage high transaction fees to gain more money than they're supposed to. We present a novel construction of payment channels that helps mitigates these types of attacks.**

I. Introduction

In recent months, cryptocurrencies has come back into the spotlight, with the total market cap rising to a peak of almost $3tn (20x in less than 2 years) as trading becomes more accessible to a larger populace through popular apps like Robinhood. Along with the increase in popularity came an increase in usage, and blockchains such as Ethereum once again became congested, partially due to users becoming interested "meme" tokens such as Doge and Shiba or owning digital assets through NFTs [1]. The increase in usage and the resulting congestion caused fees to send transactions on these blockchains to rise dramatically, leading to many users to look for ways to transact without incurring high fees.

There are two approaches to scaling a blockchain to accomodate increased usage: on-chain scaling and off-chain scaling. On-chain scaling aims to improve transaction processing capabilities through changes in the protocol such as: increasing the block size to pack more transactions per block [2]; improving the storage layer by sharding the database [3], [4]; making the communication between nodes more efficient [5]; and changing the consensus layer [6], [7]. However, on-chain scaling can lead to debate and fractures among communities [8], [9] and can take a long time to reach deployment, so off-chain scaling, which require no changes to the protocol, is the preferred solution.

State channels are one of three categories of off-chain scaling solutions for Ethereum [10]. A payment channel, an instantiation of a state channel, allows users to transact by exchanging signed messages, and only defer to transactions on-chain for disputes and withdrawals. Payment channels guarantee that parties will be able to withdraw amounts from the channel that reflect the latest message that they agreed on, a property called "no counterparty risk" [11]. This approach to scaling, however, is vulnerable to attacks when the network is congested, and can cause the counterparty risk property to be violated.

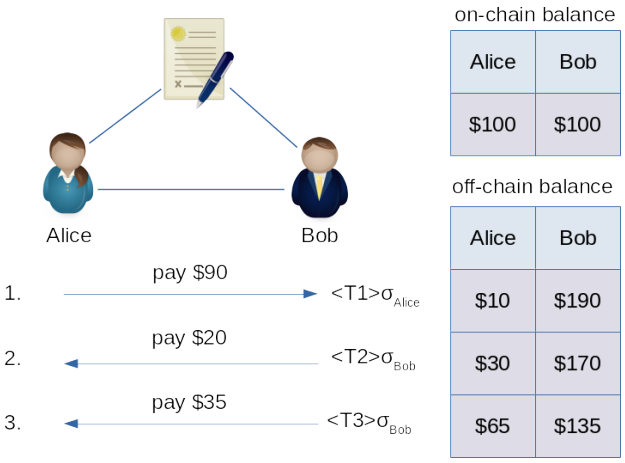

Fig. 1: A bi-directional payment channel between Alice and Bob

Consider the example shown in Figure 1. Alice and Bob set up a payment channel between each other, both committing to a deposit of $100 on the smart contract. Off-chain, both parties send a series of transactions (*T1*, *T2*, and *T3*) to each other. Under a typical network condition [12], Bob may issue a withdraw request based on off-chain state *T3* with a transaction fee of $20. And as a result, he will get $115 back to his account. Bob realizes that when the network is under a period of congestion, the transaction fee may balloon to $70. Under such a situation, if he withdraws the contract balance with the off-chain message *T1*, he will get $120 back to his account, which is $5 more comparing to playing honestly. Alice may choose to submit a dispute saying that *T3* is the latest, so that she can get $65 back. However, in order to submit this dispute, Alice has to pay $70 of transaction fee. That means Alice not only gets nothing back to her account, she will also lose another $5. If she keeps silent, she will actually get $10 back to her account when the network congestion is gone. Therefore, Alice is more likely to keep silent and take $10 back.

The action from Bob is a type of attack of payment channels called an execution fork attack [13]. Previous work have

978-1-6654-9538-7/22/$31.00 ©2022 IEEE

aimed to address one problem from these attacks, which is guaranteeing that Alice's dispute transaction makes it to the contract. We have shown with the previous example that Alice stands to lose money regardless of whether a dispute is successful, so ensuring the delivery of the dispute transaction only solves part of the problem. The smart contract needs a way to prevent Bob from making a malicious withdrawal when the network is congested, and have some way to punish Bob and compensate Alice if she is able to successfully dispute.

In this paper, we present a simple and novel redesign of smart contracts for payment channels that would help mitigate execution fork attacks. We realize that the only way to compensate Alice for transaction fees that could exceed the remainder of Bob's balance is by having a security deposit. With a deposit separate from Bob's transaction balance, Bob would forfeit his deposit to Alice if she successfully dispute his on-chain balance update and withdrawal attempt. We present a construction of payment channels that provides compensations for successful disputes when network fees are high, and provide a proof-of-concept implementation of the on-chain contract in Solidity.

II. Background and Related Work

Blockchains, smart contracts, and off-chain scaling. Leading cryptocurrencies such as Bitcoin [14] and Ethereum [15] are known to have scalability issues where the network fees explode when there is a large, sudden increase in usage [16], [17]. One of the benefits of these blockchains is that they allow for smart contracts, self-executing code that are deployed on the blockchain which users (or other contracts) can interract with. These contracts allow for applications to move data and computation off chain, afterwhich off-chain transactions can be made without incurring high on-chain transaction fees and latency. This approach to scaling blockchains is called off-chain scaling or layer-2 scaling.

State channels [11], Plasma [18], and Rollups [10] are three major off-chain scaling solutions that utilize smart contracts. Payment channels are instantiations of state channels that restricts participants to simple monetary transactions. In a payment channel, two users, Alice and Bob, who wish to transact off-chain can deploy a smart contract that contains a deposit from each of them. After creating the contract, Alice sends a message of the form (i, bal_A, bal_B, σA) to send a payment to Bob. In this message, i denotes the round number for the transaction, where a higher round number signifies a more recent transaction. Both Alice and Bob's balance after the payment are reflected by the fields bal_A and bal_B, respectively. Alice also includes a signature σA that shows her commitment to the round and user balances for that round.

If Bob wants to claim the money that is reflected by the balances in round i, he includes his own signature in the message to form (i, bal_A, bal_B, σA, σB) and send it in a transaction to the contract. The contract first verifies that the message has a higher round number than the last one it has seen, then verifies the signatures σA and σB. Upon a successful verification, the contract will update the on-chain balances to reflect the off-chain balances in round i. After this, Bob can initiate a withdrawal of the amount in his balance from the contract, but will need to wait for a predefined period of time (Δ_T). A withdrawal event will be signaled by the contract, and if there is no transaction from Alice after Δ_T has passed, Bob can withdraw his money.

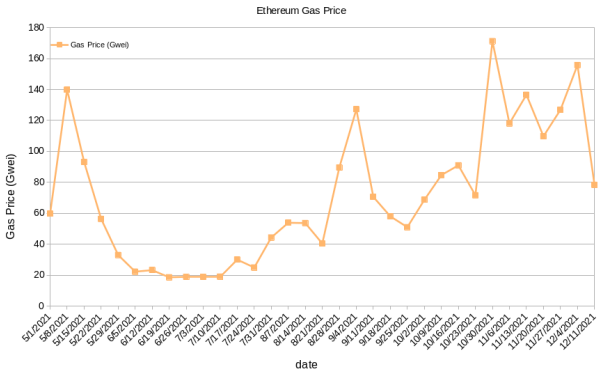

Fig. 2: Gas price to send a transaction on Ethereum over time

Impact of network congestion on payment channels. Figure 2 shows the "gas price" for Ethereum, which dictates the price of sending a transaction on chain [1]. As the network become congested, this gas price will increase, making the cost to send a transaction more expensive. Suppose that round i above was not the latest transaction between Alice and Bob, and that Bob sent Alice a large sum of money between rounds i and the most recent round $i + k$. If the on-chain state of the channel in the smart contract reflects transaction i, and Bob initiates a withdrawal of his on-chain balance, Bob could cheat Alice out of the money he sent her in last k rounds. In order to not lose money, Alice needs to refute Bob's withdrawal by sending a *dispute* transaction that contains the off-chain state for round $i + k$ before the challenge period Δ_T ends. However, if the cost to send transactions is too high due to a congested network, Alice might elect to send a transaction with a low fee, which might not make it within Δ_T. After a successful withdrawal by Bob, Alice would not be able to withdraw her balance reflected by round $i + k$, even if her *dispute* transaction makes it to the contract because there is no longer sufficient funds in Bob's on-chain balance to pay Alice.

Mitigation strategies. To mitigate execution fork attacks, payment channels must ensure that valid dispute transactions make it to the contract before the challenge period ends. One way to do this is to set a large Δ_T, which would give the network more time to decongest and for Alice's low-fee transaction to be included in a block. While a large Δ_T would work for brief periods of congestion, Ethereum and Bitcoin has both experienced prolonged network congestion that lasted for months on end. Setting a Δ_T that encompasses even these long periods of congestion is not practical, because it would cause funds to be locked up for too long, even if the party making

[1]Data retrieved from Etherscan [19]

the withdrawal is honest.

Another challenge in mitigating execution forks is ensuring that honest parties are online to be able to dispute; if Alice is offline during the challenge period, she cannot dispute even if she is willing to pay a high transaction fee. Previous works proposed appointing a custodian that could help dispute Bob's withdrawal on Alice's behalf if she is offline [13], [20], [21]. When Alice receives the payment from Bob for round *i*+*k*, she can sign and relay the message to the custodian who constantly monitors the blockchain for messages to the contract. The custodian can thus send a transaction on Alice's behalf if it sees that Bob is attempting an execution fork attack. These solutions still do not address the issue of high fees during congestion. If Bob's off-chain balance is zero after round *i*+*k*, a successful dispute still causes the custodian to lose fees for the transaction. The only way to reimburse Alice (or the custodian) for the dispute is to have a separate balance for Bob that is untouched by their off-chain transactions, and this deposit can then be used to cover the cost of the dispute transaction if it is successful. Our work realizes this missing part in the solutions to execution fork attacks, and aims to provide a better construction of payment channel contracts to ensure fairness for both parties in the presence of high fees.

III. Building Robust Payment Channels

A. Contracts with Deposits and Reactive Withdrawal Period

1) Construction: Figure 3 shows our construction of a bi-directional payment channel. When Alice and Bob wants to form a payment channel, they first need to reach agreement on the conditions of the channel: initial balances and time to wait before they can withdraw in the number of blocks. After agreeing on the terms, they exchange signatures on this message and send a transaction to create this contract. the function **constructor** will initalize the contract based on the agreed-upon terms. A deadline is set to infinity, because it only starts when one party signals the intention to withdraw. The contract also records the party that closes the channel and punished.

2) Balance Update: The payments that are sent by Alice and Bob are only reflected in their off-chain state. To sync up the on-chain state with the off-chain state, either party can call the **update** function, which will assign the on-chain balances with the off-chain balances sent in the message if all the signatures are valid. It is required that the update must be using a state that is more recent by that in the smart contract, and the contract enforces this by checking the round number.

3) Withdraw: To close the channel and withdraw money, either party can call the **withdrawExit** function. They will also include their off-chain state of the channel, which will be used to update the on-chain state. The smart contract will signal an event to notify the other party that the channel is being closed to withdraw money. At this point, the network could be under heavy network congestion, and the closing party could be malicious and try to withdraw from the channel using an old off-chain state that benefits them. The closing party is required to include a *deposit*, which will be held by the smart contract, and returned back to the closing party if there is no dispute or used to compensate the other party if there is a successful dispute. If the other party has no objection before the deadline, the contract will be closed by returning the remaining balance to both parties and the withdraw deposit to the closing party.

```
on constructor(bals, deps, withdrawwait)
    balances := bal
    Δ_T := withdrawwait
    deadline := ∞
    round := 0
    deposit := ∅
    closingparty := ∅
    punished := false

on update(bals, _round) by sender
    require _round > round
    for i := 0 to bals.len
        if bals[i]! = balances[i] and
            closingparty ! = ∅ and
            punished := false
                balances[¬closingparty] += deposit
                punished := true
        balances[i] = bals[i]
    round := _round

on withdrawExit(_bals, _round, _deposit) by sender
    if closingparty == ∅ then
        update(_bals, _round)
        deposit := _deposit
        deadline := currentblock + Δ_T
        closingparty := sender
        emit(withdrawExit, sender)

on withdraw(bals) by sender
    require currentblock > deadline
    amount := balances[sender]
    if sender == closingparty and !punished
        amount+ = deposit
    send amount to sender
```

Fig. 3: Smart contract for bi-directional payment channels.

In section IV-A, we discuss how we utilize a new functionality provided by Ethereum smart contracts, which allows the contract to view the block's base fee. To withdraw the balance, the requesting party not only needs to pay the transaction fee to submit the request, but also needs to make a temporary deposit to the contract that has the same amount as the fee to send a dispute transaction. By using the same amount of transaction fee of the dispute request, we can have an accurate estimate about how much it will cost to submit a dispute request under current condition.

4) Dispute: If the contract is about to be closed with a older state, the other party may submit a dispute to update the latest state. In the traditional smart contract, under a congestion situation, the disputing party have risk of losing money as explained in section I. With the dispute deposit, the disputing party no longer needs to worry about losing the

money, because once the dispute is approved by the smart contract, the withdraw deposit will be paid to the disputing party in order to cover the cost of submitting a dispute to the smart contract.

The **update** function also serves as a dispute resolution mechanism. If one of the parties tries to close the channel and the other party submits a state that is more recent, then the closing party will be assumed to be malicious and their deposit amount will be forfeited.

B. Withdrawal Deposit

Our design withholds the deposit in the minimum amount and in the shortest time. By requesting the withdraw deposit to have the same amount as the cost to submit request, we can avoid both cases of undercharge (if the cost for submitting a dispute is actually higher) and overcharge (if the cost for submitting a dispute is actually lower). Compared to making additional deposit at the construction time, making the withdraw deposit with the closing request only requires the closing party (instead of both parties) to make the deposit, and the deposit is only withheld from the closing request is received to the contract is closed.

C. Payment Channel Networks

Payment channels can be used to construct payment channel networks [22]–[24], which link together bi-directional channels and allow parties to send multi-hop transactions between each other. Two parties that want to transact does not even need to have a direct channel between each other, provided that there's a route between and that each hop has sufficient funds to make the payment. These payment networks also suffer from the same vulnerability to execution fork attacks as a single payment channel. A malicious party that attempts the attack could cause a cascading effect if successful. Suppose there's a network of channels between Alice, Bob, and Trudy. If Trudy cheats Bob out of his funds, Bob might not have enough left on his balance to be able to pay Alice. In this case, it is crucial that Bob's dispute transaction makes it to the contract, which can be done by enlisting a custodian. A custodian watching the blockchain for Trudy's malicious withdrawal can submit a dispute, preventing Bob from having insufficient funds to pay Alice while being compensated for the dispute transaction from Trudy's deposit.

IV. Evaluation

A. Implementation in Ethereum

As a proof-of-concept, we implemented our payment channel contract in Solidity [25], and tested the contract on Ethereum's Rinkeby testnet. Much of the contract translates directly from the pseudocode in Figure 3; however, we highlight several important differences. Firstly, we assign the *closingparty* based on the party that sent the transaction to *withdrawExit*. Initially, this variable is set to the zero address "address(0)", and it is used to check two conditions: 1) the dispute process can only be initiated once and 2) the closing party does not try to dispute its own transaction and claim the deposit. Secondly, Solidity contracts have access to the block number in which the transaction was executed, allowing it to set a deadline that is Δ_T number of blocks from the current block. Finally, the latest London update in Ethereum allows the contract to also have access to the block's base fee, allowing it to enforce a deposit amount from the closing party that is exactly equal to the cost of sending a dispute transaction at the current point in time. An honest party looking to dispute this withdrawal thus does not have to worry about the cost of the transaction if they were to immediately send the dispute. This feature is unique to Ethereum, and it discourages a party looking to perform an execution fork attack to do so when the network fees are high. We released the code for our Solidity payment channel, which can be accessed online [26].

B. Cost of Transactions

function	gas cost	median cost $	max cost $
update	56217	17.19	28.62
dispute	76781	23.48	39.09
withdrawExit	134379	41.09	68.41
withdraw	34749	10.62	17.69

TABLE I: Cost of calling each function on Ethereum's mainnet

Given our implementation of the contract in Ethereum, we evaluate the cost of executing each of the functions. Table I shows the functions in the contract and their corresponding gas cost and cost in USD. To get the median and maximum dollar amount, we look at the base fee for the most recent 1000 blocks on the mainnet, and used the values in these blocks along with the gas cost. As expected, the cost to send these transactions on the mainnet would be very high given the current state of the network even if we're looking at the median fees. This shows that any payment channel currently deployed on the mainnet, where the difference between the on-chain and off-chain balances benefits a party within the price to dispute, are susceptible to a malicious execution fork attack.

V. Conclusion

Payment channels are one of the leading scalability solutions for Ethereum, allowing users to transact simply by signing messages, and avoids the high transaction fees and latencies. These payment channels must guarantee that there are no counterparty risks, so that a user can go on-chain to claim the amount of money that reflects their most recent off-chain transaction. Current solutions to guard against execution fork attacks, wherein a malicious user attempts to withdraw money using an old state, only solves problem of ensuring a successful dispute without any regards to high fees to send these transactions during periods of congestion. We present a novel construction of payment channels that solves this part of the problem in execution fork attacks.

References

[1] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token (nft): Overview, evaluation, opportunities and challenges," *arXiv preprint arXiv:2105.07447*, 2021.

[2] "Bitcoin cash," https://bitcoincash.org/, 2021 (accessed December 10, 2021). [Online]. Available: https://bitcoincash.org/

[3] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 95–112.

[4] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 17–30.

[5] A. P. Ozisik, G. Andresen, G. Bissias, A. Houmansadr, and B. Levine, "Graphene: A new protocol for block propagation using set reconciliation," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017, pp. 420–428.

[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[7] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[8] Y. Kwon, H. Kim, J. Shin, and Y. Kim, "Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash?" in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 935–951.

[9] L. Kiffer, D. Levin, and A. Mislove, "Stick a fork in it: Analyzing the ethereum network partition," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 94–100.

[10] V. Buterin, "An incomplete guide to rollups," https://vitalik.ca/general/2021/01/05/rollup.html, 2021 (accessed December 10, 2021). [Online]. Available: https://vitalik.ca/general/2021/01/05/rollup.html

[11] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 508–526.

[12] BitInfoCharts, "Ethereum avg transaction fee historical chart." [Online]. Available: https://bitinfocharts.com/comparison/ethereum-transactionfees.html#1y

[13] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller, "Pisa: Arbitration outsourcing for state channels," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 16–30.

[14] S. Nakamoto *et al.*, "Bitcoin," *A peer-to-peer electronic cash system*, 2008.

[15] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[16] A. Chauhan, O. P. Malviya, M. Verma, and T. S. Mor, "Blockchain and scalability," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 122–128.

[17] M. Bez, G. Fornari, and T. Vardanega, "The scalability challenge of ethereum: An initial quantitative analysis," in *2019 ieee international conference on service-oriented system engineering (sose)*. IEEE, 2019, pp. 167–176.

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

[19] "Ethereum average gas price chart," https://etherscan.io/chart/gasprice, 2021 (accessed December 10, 2021). [Online]. Available: https://etherscan.io/chart/gasprice

[20] "Private altruist watchtowers," https://github.com/lightningnetwork/lnd/blob/master/docs/watchtower.md, 2021 (accessed December 10, 2021). [Online]. Available: https://github.com/lightningnetwork/lnd/blob/master/docs/watchtower.md

[21] "Raiden monitoring service," https://raiden-network-specification.readthedocs.io/en/latest/monitoring_service.html, 2021 (accessed December 10, 2021). [Online]. Available: https://raiden-network-specification.readthedocs.io/en/latest/monitoring_service.html

[22] C. Egger, P. Moreno-Sanchez, and M. Maffei, "Atomic multi-channel updates with constant collateral in bitcoin-compatible payment-channel networks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 801–815.

[23] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.

[24] P. Li, T. Miyazaki, and W. Zhou, "Secure balance planning of off-blockchain payment channel networks," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1728–1737.

[25] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017, vol. 318.

[26] T. Tran and H. Zheng, "payment-channel-congestion," 2021. [Online]. Available: https://github.com/KohdMonkey/payment-channel-congestion/blob/main/paymentchannel.sol