Two-Party State Channels with Assertions

Chris Buckland and Patrick McCorry

Kings College London, UK
`cpbuckland88@gmail.com` `patrick.mccorry@kcl.ac.uk`

Abstract. An empirical case study to evaluate state channels as a scaling solution for cryptocurrencies demonstrated that providing an application's full state during the dispute process for a state channel is financially costly (i.e. \$0.24 to \$8.83 for a battleship game) which can hamper their real-world use. To overcome this issue, we present *State Assertion Channels*, the first state channel to guarantee an honest party is always refunded the cost if it becomes necessary to send an application's full state during the dispute process. Furthermore it ensures an honest party will pay an approximate fixed cost to continue an application's execution via the dispute process. We provide a proof of concept implementation which demonstrates it costs approximately \$0.02 to submit evidence regardless of the smart contract's application.

1 Introduction

Blockchain-based cryptocurrencies do not scale. The community is pursing three approaches to alleviate the scalability issue. These are: new blockchain protocols [1], sharding transactions into distinct processing areas and off-chain protocols. While the first two approaches can strictly increase the network's throughput, they harm the network's public verifiability as they reduce the diversity of peers with the computatonal, bandwidth or storage requirements to validate all transactions on the network and ultimately hold the miner's accountable. This paper focuses on the off-chain (or so-called Layer 2) approach that simply aims to reduce the network's load. One promiment off-chain approach are state channels that lets a group of parties process transactions (and execute a smart contract) locally amongst themselves instead of on the global network. In the best case, the application is no longer restricted by the underlying blockchain's latency and all execution is free as it remains local between the parties. If there is a disagreement about the latest state of the smart contact, then any party can trigger a dispute and rely on the underlying blockchain to arbitrate the dispute's outcome. To arbitrate, the blockchain provides a fixed time period to collect evidence from all online parties before using this evidence to decide the off-chain smart contract's new state. So far, there are two types of dispute processes for a state channel. The first is called *closure dispute* [3,7,9] as the dispute process is responsible for closing the channel, re-deploying the smart contract with the new state and letting parties continue its execution via the blockchain. The second is called *command-issuance dispute* [8,6,5,2] as the dispute process collects commands from each party and then executes the command to compute the new state. A recent case study empirically evaluated state channels as a scaling solution by building the two player game battleship [7]. They highlighted the most expensive aspect of any dispute process is sending the application's full state. For example, the case study claimed that sending the full game state approximately costs \$0.24 when the blockchain is not congested, but it can potentially sky-rocket to \$8.83 if the blockchain is congested. The above can clearly hamper real-world use of state channels as an honest party will not use the dispute process if it is too clostly (and thus they cannot self-enforce the application's correct execution). The case study also highlighted that as the application cannot be progressed withing the channel then, for liveness to be preserved, they would have to made on-chain. This could be expensive if the application required computationally complex progressions or if the number of transactions was high. We propose, *State Assertion Channels*, which has a command-issuance dispute process[1]. and overcomes the issue of sending the application's full state by leveraging the concept of an optimistic contract. Our contributions:

– We propose the first state channel that ensures an honest party never pays the cost to send the application's full state during the dispute process.
– Our state channel is also the first to ensure the cost of progressing an application is based only on the number of transactions required to reach a terminal state, thus it is independent of the application's computational cost.
– We provide a proof of concept implementation and experimentally demonstrate that it is cost-effective to deploy.

2 Background

Optimistic Smart Contracts An optimistic smart contract trades the cost of computation for time. This lets a smart contract accept an application's new state if no one has proved it is invalid within a fixed challenge period. Briefly, one party submits to the optimistic smart contract the application's $\mathsf{state_i}$, a command cmd, its inputs, the next $\mathsf{state_{i+1}}$ and a financial bond. This asserts that $\mathsf{state_{i+1}}$ is the next state if the smart contract were to compute $\mathsf{state_{i+1}} = \mathtt{Transition}(\mathsf{state_i}, \mathsf{cmd}, \mathsf{inputs})$. Other interested parties can compute the transition locally to verify its validity. If the asserted state is invalid, then anyone can issue a challenge by notifying the smart contract to compute the transition. If the challenge is successful, then a bond is used to refund the challenger. Eigenmann, Moore and Johnson provided the first demo implementation of an optimistic contract for the Ethereum Name Service [4], but so far this technique has alluded real-world use.

Command Issuance State Channels Sprites proposed the concept of a command-issuance state channel, and since then it has been extended by PISA [6], Counterfactual [5] and Magmo [2]. At a high level, one party can submit the latest

[1] The assertion concept isn't compatible with closure channels as the full state is required to re-deploy an application.

$\mathsf{state_i}$ agreed by all parties before triggering the dispute process. The smart contract provides a fixed time period for all parties to submit commands and the contract is responsible for computing every command (i.e. state transition). In Sprites (and PISA), all commands are executed after the dispute process has expired. Whereas in Counterfactual and Magmo, the command is executed when it is submited and the dispute period's expiry time is reset (i.e. it dispute period is extended for every command). The dispute process can be cancelled if one party submits a later state agreed by all parties, or after the expiry time.

3 State Assertion Channels

The state assertion contract **SC** and the application contract **AC** must be deployed on the blockchain. Each party must lock coins into the state assertion contract before the state channel is activated. Both parties can co-operatively execute the application off-chain amongst themselves by executing every state transition for **AC** locally and exchanging signatures for every new state. If there is a disagreement off-chain, then both parties can continue its progression via the dispute process.

Our contribution involves changing the dispute process to avoid sending the full application's state, and to avoid computing the next states on-chain. Instead parties will assert an application's new state by submitting a hash of the previous state $\mathsf{hstate_i}$, the command cmd, its inputs, a hash of the next state $\mathsf{hstate_{i+1}}$ and a financial bond. The dispute process provides a fixed time period for the counterparty to verify the assertion by computing the state transition locally. To challenge an assertion, the party submits the previous $\mathsf{state_{i-1}}$ which lets **SC** verify the assertion was indeed correct by executing the transition via **AC**. If the honest party successfully challenges an assertion and proves it is invalid, then they are sent the bond as a refund.

Thus an honest party can always continue an application's execution via the blockchain's dispute process by asserting a hash of the next state. As well, they are always refunded the cost of sending the application's full state in order to challenge an invalid state assertion.

3.1 Application Contract Assumptions

Turn-based application We assume that the parties execute a turn-based application where each party performs their state transition in turn until a terminal state is reached. As well, **AC** must be instantiated on the blockchain to ensure its address is provided to the state assertion contract **SC**.

Single transition function The application contract implements a transition function which accepts the full state, a command and a list of inputs. The application contract is responsible for computing a state transition and returning a hash of the new state via $\mathsf{hstate_{i+1}} := \mathtt{Transition}(\mathsf{state_i}, \mathsf{cmd}, \mathsf{inputs});$. The application contract is stateless consequently the state must be supplied to compute a state transition.

No exceptions or out-of-gas errors In Ethereum, an entire transaction's execution can be reverted if a smart contract throws an exception (i.e. out of gas). If the application **AC** can throw an exception, then it can be used to revert the execution of an honest party's challenge. Thus the application's transition function should not permit exceptions. We propose the transition function should return hstate and if the command doesn't exist or its execution simply fails, then it should return $\mathsf{hstate} = 0$. This ensures an honest party can always issue a challenge via **SC**.**challenge**() as the state transition (and the verification) will always complete its execution.

3.2 Assertion Channel Overview

Figure 1 presents the state channel assertion contract. We'll use it to aid the following overview on how to instantiate the contract, authorise states off-chain, how to trigger a dispute, how to submit and challenge assertions, and finally how to close the channel.

Channel status The channel has three flags $\texttt{Status} = \{\texttt{DEPOSIT}, \texttt{ON}, \texttt{DISPUTE}\}$. Both parties must deposit coins in **SC** before it will transition from $\texttt{DEPOSIT} \rightarrow \texttt{ON}$. While the channel's status is set as $\texttt{ON}$ both parties can co-operatively continue the application's progression off-chain by exchanging signatures for new states. If there is a disagreement about a state transition, then one party can trigger a dispute which changes the status from $\texttt{ON} \rightarrow \texttt{DISPUTE}$.

Instantiating contract One party must deploy **SC** to the blockchain and initalise it with the address of both parties $\mathcal{P}_1, \mathcal{P}_2$, the application's address **AC**, the fixed dispute period Δ and the required security bond bond. Both parties must review the contracts **SC**, **AC** and the intialisation values before sending their deposit via **SC**.**deposit**(). After **SC** has received both deposits (and before turning on the channel), it will compute the initial state $\mathsf{state}_{\texttt{initial}} = (\perp, \texttt{balance1}, \texttt{balance2})$ and declare the first turn will be taken by $\mathcal{P}_1$.[2]

Progressing application off-chain Both parties can begin exchanging signatures to execute the application off-chain when the channel is $\texttt{ON}$. In each round, one party is responsible for proposing a state transition, and the other party is responsible for verifying the state transition before co-operatively authorising it. To propose, the party computes $\mathsf{state}_{\mathsf{i}+1} = \texttt{Transition}(\mathsf{state}_{\mathsf{i}}, \mathsf{inputs}, \mathsf{cmd})$, they hash the state $\mathsf{hstate}_{\mathsf{i}+1} = \texttt{hash}(\mathsf{state}_{\mathsf{i}+1})$ and they sign its hash $\sigma_{\mathcal{P}_1} = \mathsf{Sign}(\mathsf{hstate}_{\mathsf{i}+1}, \mathsf{i}+1, \textbf{SC}, \mathcal{P}_{\texttt{turn}})$, where $\mathcal{P}_{\texttt{turn}}$ specifies the next party's turn. The proposer must send $\mathsf{hstate}_{\mathsf{i}+1}, \mathsf{i}+1, \sigma_{\mathcal{P}}$ to the counterparty. To verify, the counterparty computes state transition and the state hash $\mathsf{hstate}'_{\mathsf{i}+1}$ before checking if $\mathsf{hstate}'_{\mathsf{i}+1} == \mathsf{hstate}_{\mathsf{i}+1}$. If this condition is satisified (and $\mathsf{i}+1$ is the largest counter so far), then the counterparty signs $\sigma_{\mathcal{P}_2} = \mathsf{Sign}(\mathsf{hstate}_{\mathsf{i}+1}, \mathsf{i}{+}1, \textbf{SC}, \mathcal{P}_{\texttt{turn}})$ and sends their signature $\sigma_{\mathcal{P}_1}$ to the proposer.

[2] We highlight a subtle difference between the initial state $(\perp, \texttt{balance1}, \texttt{balance2})$ and the terminal state $(\texttt{balance1}, \texttt{balance2})$.

Triggering a dispute In general, a dispute must be triggered if the counterparty stops responding in the state channel (i.e. they do not agree with the state update and they refuse to sign it). There are two cases to consider. Either the proposer is waiting on a signature from the verifier to authorise the new state, or the verifier is waiting on the proposer to propose a new state transition. In both cases, each party waits for a local time-out before submiting the most recently $\mathsf{hstate}_\mathsf{i}$ via **SC.setstate**() and triggering a dispute via **SC.triggerDispute**(). The signed state hash includes $\mathcal{P}_{\mathtt{turn}}$ and thus **SC** waits for a new state assertion from the named party before $\mathtt{deadline} = \mathtt{now} + \Delta$. To continue off-chain and cancel the dispute, one party must submit a co-operatively signed hstate (with a larger counter i) via **SC.setstate**.

Submitting a state assertion The named party $\mathcal{P}_{\mathtt{turn}}$ must send an asserted $\mathsf{hstate}_{\mathsf{i}+1}$, the command cmd and its inputs inputs using **SC.assertState**() before the dispute process expiry time $\mathtt{deadline}$. Every time a state assertion $\mathsf{hstate}_{\mathsf{i}+1}$ is submitted, the contract resets the deadline $\mathtt{deadline} = \mathtt{now} + \Delta$ and stores the previous state assertion $\mathsf{hstate}_\mathsf{i}$ as accepted. Furthermore the contract records that it is the counterparty's turn to respond. In terms of the financial bond, the contract only needs to store a single bond per party which can be collected when the party asserts a new state or when the parties send their initial deposit.

Responding to a state assertion The counterparty is responsible for verifying if a state assertion is correct by computing $\mathsf{state}_{\mathsf{i}+1} = \mathtt{Transition}(\mathsf{state}_\mathsf{i}, \mathsf{cmd}, \mathsf{inputs})$ locally and checking if the asserted $\mathsf{hstate}_{\mathsf{i}+1}$ represents $\mathsf{state}_{\mathsf{i}+1}$. If the state assertion is valid, then the counterparty can continue the application's execution by responding with a new state assertion using **SC.assertState**(). By continuing the application's execution, the counterparty is agreeing that the previous state assertion is valid. If the state assertion is invalid, then the counterparty can challenge it by supplying the plaintext state $\mathsf{state}_\mathsf{i}$ to the contract using **SC.challenge**(). The contract will compute the transition and confirm if $\mathsf{hstate}_{\mathsf{i}+1}$ represents the new state $\mathsf{state}_{\mathsf{i}+1}$. If the challenger is successful and proves the state assertion as invalid, they are sent all coins in the channel (including the counterparty's bond to refund the cost of this transaction).

Reaching the terminal state In Section 3.1, we assumed an application's execution will always reach a terminal state which is simply the final balance of both parties $\mathsf{state}_{\mathtt{final}} = (\mathsf{balance1}, \mathsf{balance2})$. The final $\mathsf{hstate}_{\mathtt{final}}$ must be accepted by the assertion contract **SC** before both parties are sent their final balance by supplying $\mathsf{state}_{\mathtt{final}}$ to **SC.resolve**(). It is clear if both parties continue the application's execution co-operatively off-chain, then they can simply send the terminal state hash via **SC.setstate**() before resolving the channel. On the other hand, the dispute process enforces turn-based state assertions to ensure that one party will eventually propose the terminal state hash via **SC.assertState**(). When the terminal state hash is reached, the counterparty's only option is to submit $\mathsf{state}_{\mathtt{final}}$ before the deadline using **SC.resolve**()

4 Discussion and Future Work

Proof of concept implementation We developed a proof of concept for the Ethereum blockchain. Our smart contract is written in Solidity[3], and gas costs were measured using a private network. The assertions contract costs 2,943,664 gas to deploy, approximately \$0.97 using the gas price of 2.6 Gwei and the conversion rate of 1 ether = \$127 which was the real world rate in January 2019. The cost to make a state assertion is only 59,774+39.5n gas (\$0.02 at 2.6 Gwei and \$0.77 on a congested network at 96 Gwei) where n corresponds to the number of bytes supplied as inputs to the assertion. Compared to the 725,508 gas (\$0.24 at 2.6 Gwei and \$8.83 at 96 Gwei) required to send the full battleship state.

Honest party can always verify state assertions To issue a challenge or continue the application's execution, an honest party must have the $\mathsf{state}_\mathsf{i}$ which corresponding to the contract's accepted $\mathsf{hstate}_\mathsf{i}$. There are only two situations when a new $\mathsf{hstate}_\mathsf{i}$ can be accepted by **SC**. In the first situation, $\mathsf{hstate}_\mathsf{i}$ will be accepted by **SC** if it is submitted using **SC**.**setstate**(), but this requires both parties to have already signed it (and thus acknowledge they know $\mathsf{state}_\mathsf{i}$). In the second situation, a new $\mathsf{hstate}_\mathsf{i}$ will be accepted by **SC** if the counterparty has asserted it using **SC**.**assertState**() and if the honest party continues the application's execution by asserting the next $\mathsf{hstate}_{\mathsf{i}+1}$ via **SC**.**assertState**(). We highlight the contract accepts $\mathsf{hstate}_\mathsf{i}$ as the honest party has countinued its execution instead of challenging it. As the above demonstrates, an honest party will always have a copy of $\mathsf{state}_\mathsf{i}$ if the corresponding $\mathsf{hstate}_\mathsf{i}$ is accepted by the contract. Thus they can always verify state transitions and issue challenges.

Motivation for turn-based commands There are two motivations for the turn-based channel. First each party can submit a state assertion and the counterparty is always provided an opportunity to accept or challenge it. Second, each state assertion must strictly build upon a previously accepted state hash. If there are two or more state assertions that reference the same previous state hash, then **SC** can only accept one state assertion. Because of the requirement to strictly order state assertions and the need to 'accept the first received state assertion', this lets an attacker simply pay a higher fee and front-run an honest party to ensure their state assertion is always accepted first (i.e. front-running ensures an honest party's state assertion is never accepted by **SC**). Thus the turn-based nature of this state channel prevents the above front-running attack.

Enforcing time-based events The assertion channel is responsible for enforcing time-based events with the dispute period Δ. When the application is co-operatively progressing off-chain, an honest party will wait for a local timeout before triggering a dispute via the blockchain. For every new state assertion, the dispute process is reset to ensure each party has a time period of Δ to take their next move. If a party doesn't assert a new state before the deadline, then

[3] https://pastebin.com/UBVvZ0FU

the honest party will notify the contract via **SC.timeout**(). This terminates the application and sends all coins (including the bonds) to the honest party.

Bond Requirement Each party must deposit a bond to cover the cost of a successful challenge to their assertion. A bond's value must consider the worst-case when a transaction fee spikes due to network congestion. For example, in the battleship empirical case study it was highlighted that submitting the game's state can sky rocket from \$0.24 to approximately \$8.83 during network congestion. If the security bond isn't sufficient to challenge a state assertion, then the counterparty may not challenge it.

Offline parties and PISA If the honest party is offline, then the counterparty can trigger a dispute with the latest agreed hash and then assert an invalid state hash (i.e. sends the counterparty all the coins in the channel). If the offline party relies on a watching service, like PISA [6], then the watching service must have a copy of the latest state in plaintext to verify the invalid state transition and issue a challenge. However this hinders state privacy as the watching service can view the channel's internal state. As well, a watching service cannot perform a valid state transition on the offline party's behalf, so the offline party must ensure the only valid state transition for the counterparty is the application's terminal state.

Extending to N-parties Future work should investigate how the state assertion paradigm could be extended to n-party state channels. Channels could progress in a round-robin fashion and store the last **n** state assertions. This is so that a party can be sure that a state assertion will not be accepted unless they explicitly apply their own state assertion after it.

References

1. Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.
2. Tom Close and Andrew Stewart. Force-move games, 2018. `https://magmo.com/force-move-games.pdf`.
3. Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. Cryptology ePrint Archive, Report 2018/320, 2018. `https://eprint.iacr.org/2018/320`.
4. Dean Eigenmann. Optimistic contracts. Accessed 10/01/2019, https://medium.com/@decanus/optimistic-contracts-fb75efa7ca84.
5. Liam Horne Jeff Coleman and Li Xuanji. Counterfactual: Generalized state channels, 2018. `https://l4.ventures/papers/statechannels.pdf`.
6. Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. *IACR Cryptology ePrint Archive*, 2018:582, 2018.
7. Patrick McCorry, Chris Buckland, Surya Bakshi, Karl Wüst, and Andrew Miller. You sank my battleship! a case study to evaluate state channels as a scaling solution for cryptocurrencies.

8. Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Suryai Bakshi, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*, 2017.

9. ScaleSphere Foundation Ltd. ("Foundation"). Celer network: Bring internet scale to every blockchain. Technical report. Accessed 10/01/2019, https://www.celer.network/doc/CelerNetwork-Whitepaper.pdf.

```
contract StateAssertionChannel {
  enum Status {DEPOSIT, ON, DISPUTE}, address[] plist;
  Status status; uint deadline; hash hstate_i, uint i; address turn;
  address asserter; hash hstate_{i+1}; bytes input; uint cmd; bool assertion;

  function setstate(bytes[] _sigs, uint _i, address _turn, hash _hstate_i) {
    require(_i > i)); // Largest counter so far
    hash hmsg = hash(_i, _hstate_i,_turn, address(this));
    require(verifySigs(hmsg, sigs, plist)); // Everyone signed new hstate
    delete(input,cmd,hstate_{i+1},asserter, assertion); // Delete assertion
    status = Status.ON; i = _i; hstate_i = _hstate_i; turn = _turn; // Agreed state
  }

  function triggerDispute() {
    require(status == Status.ON & onlyParties()); // Only parties trigger
    status = Status.DISPUTE; deadline = now + disputePeriod;
  }

  function assertState(hash _hstate_i, hash _hstate_{i+1}, bytes _input, uint _cmd) {
    require(status == Status.DISPUTE AND checkCallerTurn());
    if(!assertion) {
      assertion = true; require(hstate_i == _hstate_i) // First assertion
    } else { require(hstate_{i+1} == _hstate_i); } // Extending existing assertion
    asserter = msg.sender; input = _input; cmd = _cmd; // Store assertion
    hstate_i = hstate_{i+1}; hstate_{i+1} = _hstate_{i+1}; // i accepted. i+1 assertion
    deadline = now + disputePeriod; // Reset deadline after an assertion
    progressTurn()); // Increment the turn counter
  }

  function challengeAssertion(bytes _oldstate) {
    require(status == Status.DISPUTE AND checkCallerTurn());
    require(hstate_i == hash(_oldstate) AND assertion); // Assertion exists
    hash check = AC.transition(asserter, _oldstate, input, cmd); // Compute
    if(hstate_{i+1} != checkh) { // Send all coins and bond to non-cheater. }
  }

  function timeOut() {
    require(now >= deadline && status == Status.dispute);
    // Send all coins/bonds to asserter (i.e. last party to respond).
  }

  function resolve(uint balance1, uint balance2) {
    if(status == Status.Dispute) {
      require(checkCallerTurn()); // Non-asserter must resolve b4 timeout.
    } else { require(status == Status.ON); } // No on-going dispute.
    require(hstate == hash(balance1, balance2)); // Terminal state?
    // Send each party their final balance and bond.
  }
  function deposit(); // Not implemented due to space - INCLUDES BOND
  function onlyParties() returns(bool); // Check if tx signer is whitelisted
  function refundAllBonds() internal; // Refunds all bonds
  function checkCallerTurn() returns(bool); Enforce turn based disputes
  function progressTurn() returns(bool); Update the turn counter
  function verifySigs(bytes hmsg, bytes[] sigs, address[] signers) returns(bool);
}
```

Fig. 1: Example of the state assertion contract