You sank my battleship! A case study to evaluate state channels as a scaling solution for cryptocurrencies

Patrick McCorry[1], Chris Buckland[1], Surya Bakshi[2], Karl Wüst[3], and Andrew Miller[2]

[1] King's College London UK
patrick.mccorry,chris.buckland@kcl.ac.uk
[2] University of Illinois at Urbana Champaign
sbakshi3,soc1024@illinois.edu
[3] ETH Zurich
karl.wuest@inf.ethz.ch

Abstract. Off-chain protocols (or so-called Layer 2) are heralded as a scaling solution for cryptocurrencies. One prominent approach is called a state channel which allows a group of parties to transact amongst themselves and the global blockchain is only used as a last resort to self-enforce any disputed transactions. To evaluate state channels as a scaling solution, we provide a proof of concept implementation for a two-player battleship game. We built battleship as it fits the category of applications (i.e. chess, tic tac toe, poker, etc) that are not reasonable to execute on the blockchain directly, but it is perceived as an ideal application for a state channel. We explore the minimal modifications required to deploy the battleship game as a state channel and propose a new state channel construction, Kitsune, which combines features from existing constructions. While in the optimistic case we demonstrate the battleship game can be played efficiently in a state channel, the requirement for all parties to collectively authorise new transactions in the state channel introduces new economic and time-based attacks that if exploited renders the game as unreasonable to play.

1 Introduction

Since 2009, we have witnessed the rise of cryptocurrencies as the market capitalisation for all cryptocurrencies peaked to $1 trillion US dollars in December 2017. While Bitcoin was the first cryptocurrency designed to support financial transactions, another promiment cryptocurrency called Ethereum has emerged for executing programs called smart contracts. The promise of smart contracts is to support the execution of applications without human oversight or a central operator. Some applications proposed include decentralised (and non-custodial) token exchanges, publicly verifiable gambling games without dealers, auctions for digital goods without auctioneers, boardroom electronic voting wthout tallying authorities, etc.

Cryptocurrencies do not yet scale. Bitcoin can support approximately 7 transactions per second and Ethereum can support around 13 transactions per second. The lack of scalability is one of the primary hurdles preventing global adoption of cryptocurrencies as the network's transaction fee typically become unaffordable for most users whenever the transaction throughput ceiling is reached (i.e. the average fee in Bitcoin reached \$20 in December 2017). The community is pursuing three approaches to scale the network which include new blockchain protocols, sharding the blockchain and off-chain protocols. New blockchain protocols can strictly increase the network's throughput [26,13,27], whereas sharding can be used to distribute transactions into processing areas such that peers only validate transactions that interest them [18,1,20]. However there is a tradeoff between increasing the network's transaction throughput to support a larger userbase in terms of affordable fees, and the number of validators with the necessary computational resources to validate every transaction [16].

An alternative scaling approach consists of off-chain solutions to reduce the number of transactions processed by the blockchain. It lets a group of parties deposit coins in the blockchain for use within an off-chain application. Afterwards all parties can transact amongst themselves without interacting with the global network and the deposited coins are re-distributed depending on the application's outcome. Two proposals include an alternative blockchain (i.e. a sidechain) or a channel. A sidechain has block producers (i.e. miners or a single operator) for deciding the order of transactions and users who publish transactions for inclusion. There are several sidechain protocols [2,9] which bootstrap from Bitcoin (including a live network by RSK), whereas Plasma[23] and NOCUST[17] are non-custodial sidechains which bootstrap from Ethereum for financial transactions. While sidechains are a promising off-chain solution, they still require a blockchain protocol which has a transaction throughput ceiling.

On the other hand, a channel can be considered an n of n consensus protocol as all parties collectively authorise the state of an application amongst themselves. There is no blockchain protocol and all parties only store the most recently authorised state of the application. Channels first emerged in Bitcoin to support one-way payments between two parties [28,8], but has since evolved in Bitcoin towards the development of an off-chain payment network [24] by several companies including Blockstream, LND and ACINQ. At the same time, several proposals [22,21,10,11,5,19,4] collectively extend the capability of a channel to support a group of parties to execute a smart contract (i.e. a program) amongst themselves as opposed to simply payments. A state channel promises instant finality for every transaction and no transaction fees as there is no operator to reward. Channels are also self-enforcing as each party is protected against a full collusion of all other parties and in terms of scalability the throughput is only restricted by the network latency between the parties. The Ethereum Foundation has donated over \$2.7m [14] and the Ethereum Community Fund has donated \$275k [15] to further explore state channels as a scaling solution.

In this paper, we present an empirical evaluation in the form of a case study for a single-application state channel which must be a viable scaling option before a network of state channels is conceivable. To aid this evaluation we have designed a two-player battleship game as a smart contract. An application like battleship is not typically considered viable to execute via the blockchain due to the quantity of transactions required and in our experiment we confirm this perception as the financial cost is between \$16.27 and \$24.05. However, state channels are perceived as a potential scaling solution to allow applications like battleship to be executed over the blockchain. Our contributions are as follows:

– We explore the minimal modifications required to deploy a single-application smart contract as a state channel and propose a template of modifications that can be adopted by others deploying state channels.
– We present a new state channel construction, Kitsune, which is application-agonostic, supports n parties and allows the channel to be turned off such that the application's progress can continue via the blockchain. This combines the constructions from [22], [21], [10], [6].
– We provide a proof of concept implementation to evaluate deploying applications within a state channel. This experiment highlights the worst-case scenario of state channels and how it potentially renders applications like battleship as unreasonable to deploy within a state channel.

2 Background

In this section, we provide background information about smart contracts and how the concept of a channel has evolved.

2.1 Smart contracts

A smart contract can be viewed as a trusted third party with public state. It has a unique address on the network, it is instantiated based on the code supplied at the time of its creation, and all execution can be modelled as a state machine. Every transaction executes a command and this transitions the state machine $\mathsf{state_{i+1}} = \mathsf{transition}(\mathsf{state_i}, \mathsf{cmd})$. All parties must replicate the program's entire execution in order to verify the blockchain and join the network. This mass-replication self-enforces a smart contract's correct execution and also implies that all data for the smart contract must be publicly accessible. Finally all computation by a smart contract is measured using a metric called gas and the sender of a transaction sets a desired gas price. The amount of gas used by a contract invocation multiplied by the gas price sets the transaction fee for incentivising a miner to include this transaction in their block.

2.2 Evolution of channel constructions

We present a high-level overview of a channel before exploring the evolution of channel constructions from Bitcoin for financial transactions to Ethereum for executing arbitary smart contracts.

High level overview A channel lets n parties agree, via unanimous consent, to new states that could be published to the blockchain. As a result parties can transact amongst themselves instead of interacting via the global network. To set up, each party in the group must lock coins in the underlying blockchain for the channel. Afterwards all parties collectively execute state transitions and exchange signatures to authorise every new state (i.e. the balance of all parties, the state of a smart contract, etc). If a single party does not co-operate to authorise a valid state transition, then the underlying blockchain is trusted to resolve disputed transactions and self-enforce the state transition. In the case of Bitcoin, the blockchain gurantees *the safety of coins for the online parties*, whereas in the case of a smart contract in Ethereum it also guarantees *liveness such that an application will always progress and eventually terminate.*

Payment channels Spilman proposed *replace by incentive* which is the first state replacement technique for a channel. It is designed for one-way payments from a sender to receiver [28] and the receiver is responsible for publishing the state that pays them the most coins. To support bi-directional payments, Decker proposed *replace by time lock* which decrements the channel's expiry time whenever the payment direction changes [8]. However both state replacement techniques require an expiry time which restricts the total number of transactions that can occur. Poon and Dryja proposed a third state replacement technique called *replace by revocation* for Lightning Channels [24]. It requires both parties to authorise each other's copy of the new state before sharing secrets to revoke the previously authorised state. Crucially, it introduced the concept of a dispute process where one party publishes an authorised state to close the channel and the blockchain provides a fixed dispute period for the counterparty to prove the published state is invalid. Raiden proposed the first payment channel construction for Ethereum which is effectively a pair of replace by incentive channels [25]. Unlike in Bitcoin, this construction has no expiry time and does not restrict the total number of payments within the channel, but it is still restricted to two parties and the channel's state only considers the balance of both parties.

State channels Both Sprites and Perun independently proposed a new state replacement technique called *replace-by-version* [22,10], but there is a subtle difference. Sprites introduced a *command transition state channel* which supports n parties and it always remains open. Its dispute process lets one party trigger a dispute by submitting a state, its version and a list of signatures to prove this state was authorised by every party. All parties are provided a fixed time period to submit commands and every accepted command is simply executed via the blockchain after the dispute process has expired. Perun introduced a *closure state channel* which supports 2 parties. It lets the channel close and for the application's execution to continue via the blockchain. Its dispute process can be triggered if one party submits a fully authorised state. All parties are provided a fixed time period to submit states with larger versions and after the dispute process the state with the largest version is considered the final off-chain agreed state. Pisa modified the Sprites construction such that a commitment (i.e. hash) of the new state is signed instead of the plaintext state, but the state channel is still responsible for accepting commands in plaintext. Perun and Counterfactual extend the concept of a state channel in two ways [10,5] First, they proposed the state within a channel can be organised in a hierarchy to support multiple-applications and the dispute process for one application does not impact other applications in the channel. Second, they proposed virtual channels which allow two parties without a direct and established channel to connect with each other using a network of channels. This requires all channels along the route to lock up collateral while the virtual channel is open.

3 Kitsune State Channel Construction

We propose, Kitsune, the first application-agnostic state channel construction SC. Kitsune focuses on the dispute process and it only considers the list of parties, signatures, a hash of the final state, and the version number. Like Sprites, it is designed to support n parties and follows the same dispute model of triggering a dispute, submitting evidence and then finally resolving the dispute. Like Perun, it simply focuses on deciding the final agreed off-chain state to close the channel. Finally we also propose an application template AC which will lock and unlock an application into a state channel upon the approval of all parties.

3.1 Overview of Kitsune

An overview of the state channel contract is presented in Figure 2 and the application template is presented in Figure 1. Briefly, all parties must approve to lock the application using AC.lock which disables all functionality and instantiates the state channel contract. All parties continue the application's execution off-chain by collectively signing the hash of every new state alongside an incremented version. The channel can be co-operatively turned off using SC.close, or any party can trigger the dispute process using SC.trigger. If triggered, all parties have a fixed time period to publish the state hash with the largest version using SC.setstatehash. After the dispute process has expired, any party can resolve the dispute using SC.resolve which stores the final state hash with the largest version. Any party can unlock the application by submitting the entire state in plaintext using AC.unlock. The application will hash the enite state, fetch the final state hash from the state channel contract using SC.getstatehash, and compares both hashes. If satisified, the full state is stored and all functionality in the application contract is re-enabled to permit executing it via the blockchain.

3.2 Kitsune state channel contract

We provide an overview of the state channel contract for Kitsune before discussing how to instantiate it, how parties collectively authorise new states off-chain and how the dispute process is used to confirm the final state hash.

Overview of the state channel contract The state channel can be in one of three states which are $\mathsf{status} := \{\mathsf{ON}, \mathsf{DISPUTE}, \mathsf{OFF}\}$. All parties can collectively authorise new states for the application when the state channel is set as $\mathsf{status} := \mathsf{ON}$. Any party can trigger a dispute which sets the state as $\mathsf{status} := \mathsf{DISPUTE}$ and this provides a fixed time period for all parties to submit an authorised state hash (and its corresponding version). Once the dispute is resolved or if the channel is closed co-operatively, then the state is set to $\mathsf{status} := \mathsf{OFF}$ and this determines the final state hash for the application. If the channel is closed due to the dispute process, then a dispute record is stored which includes the starting time and finishing time for the dispute $\mathsf{t}_{\mathsf{start}}, \mathsf{t}_{\mathsf{end}}$ and the final version i.

Creating the channel The application contract AC is responsible for instantiating the state channel contract with the list of participants $\mathcal{P}_1, ..., \mathcal{P}_n$ and the dispute timer $\Delta_{\mathsf{dispute}}$. The state channel is set as $\mathsf{status} := \mathsf{ON}$ and the application contract's functionality is disabled.

Authorising off-chain state hashes A command cmd is a function call within the application contract. Any party $\mathcal{P}$ can select a command cmd and propose a new state transition $\mathsf{state}_{\mathsf{i}+1} := \mathsf{transition}(\mathsf{state}_{\mathsf{i}}, \mathsf{cmd})$. The new state is hashed with a blinding nonce[4] $\mathsf{hstate}_{\mathsf{i}+1} := \mathsf{H}(\mathsf{state}_{\mathsf{i}+1}, \mathsf{r}_{\mathsf{i}+1})$ and signed $\sigma_{\mathcal{P}} := \mathsf{Sign}(\mathsf{hstate}_{\mathsf{i}+1}, \mathsf{i}+1)$. To complete the state transition, the party sends $\mathsf{cmd}, \mathsf{hstate}_{\mathsf{i}+1}, \mathsf{state}_{\mathsf{i}+1}, \mathsf{r}_{\mathsf{i}+1}$ and $\sigma_{\mathcal{P}}$ to all other parties for their approval. All other parties in the channel verify the state transition before authorising it. To verify, each party re-computes the transition $\mathsf{state}'_{\mathsf{i}+1} := \mathsf{transition}(\mathsf{state}_{\mathsf{i}}, \mathsf{cmd})$ and state hash $\mathsf{hstate}'_{\mathsf{i}+1} := \mathsf{H}(\mathsf{state}'_{\mathsf{i}+1}, \mathsf{r}_{\mathsf{i}+1})$. Then each party verifies the signature $\mathsf{VerifySig}(\mathcal{P}, (\mathsf{hstate}'_{\mathsf{i}+1}, \mathsf{i}+1), \sigma_{\mathcal{P}})$ and that the version is the largest received so far. If satisfied, each party signs the state hash $\sigma_k := \mathsf{Sign}(\mathsf{hstate}_{\mathsf{i}+1}, \mathsf{i}+1, \mathsf{SC}, \mathsf{AC})$ and sends this signature to all other parties. A new state hash is only considered valid when each party has received a signature from every other party. If one party does not receive all signatures by a local time-out, then this party can trigger the dispute process to turn off the channel, unlock the application and continue its execution via the blockchain.

Dispute process Any party can trigger the dispute process using $\mathsf{SC.trigger}$. This self-enforces the dispute time period $\mathsf{t}_{\mathsf{start}} := \mathsf{t}_{\mathsf{now}}$, $\mathsf{t}_{\mathsf{end}} := \mathsf{t}_{\mathsf{now}} + \Delta_{\mathsf{dispute}}$ and sets $\mathsf{status} := \mathsf{DISPUTE}$. All parties can submit the latest state hash, its version and the list of signatures to prove it was authorised using $\mathsf{SC.setstatehash}$. The state channel contract SC only stores $\mathsf{hstate}_{\mathsf{i}}$ if it is signed by all parties and it has the largest version i received so far. After the dispute period has expired, any party can resolve it using $\mathsf{SC.resolve}$. This sets $\mathsf{status} := \mathsf{OFF}$, stores a dispute record $(\mathsf{t}_{\mathsf{start}}, \mathsf{t}_{\mathsf{end}}, \mathsf{i})$ and allows the application contract AC to fetch the final state hash $\mathsf{hstate}_{\mathsf{i}}$.

[4] The blinding nonce is used for state privacy if resolving disputes is outsourced to an accountable third party as proposed by Pisa [21]

Co-operative close All parties can sign $\sigma_{\mathcal{P}} := \mathsf{Sign}_{\mathcal{P}}('\mathsf{close}', \mathsf{hstate_i}, \mathsf{i}, \mathsf{SC})$ and submit it to the state channel using $\mathsf{SC.close}$. This stores the state hash $\mathsf{hstate_i}$, its version i and sets $\mathsf{status} := \mathsf{OFF}$. No dispute is recorded in the contract.

3.3 Application Contract Template

We present an application template that can be applied to easily add state channel support to an existing smart contract. It demonstrates how to lock all functionality in the application for use in the state channel and how to unlock all functionality to permit the application's execution to continue via the blockchain.

Overview of template. After modifications, the application contract must explicitly record a list of participants $\mathcal{P}_1, ..., \mathcal{P}_n$, a dispute timer $\Delta_{\mathsf{dispute}}$, whether the state channel has been instantiated $\mathsf{instantiated} := \{\mathsf{YES}, \mathsf{NO}\}$ and if so it also stores the state channel's address SC. All functions within the application require a new pre-condition to check whether the state channel is instantiated and should only permit execution if $\mathsf{instantiated} = \mathsf{NO}$. Finally the application must include two new functions $\mathsf{AC.lock}$ that instantiates the state channel upon approval of all parties and $\mathsf{AC.unlock}$ that verifies a copy of the full state before re-enabling the application.

Lock application contract All parties must agree to create the state channel by signing $(\mathsf{ON}, \mathsf{AC}, \Delta_{\mathsf{dispute}}, \mathsf{lockno})$, where ON signals turning on the channel, lockno is an incremented counter to ensure freshness of the signed message and $\Delta_{\mathsf{dispute}}$ is the fixed time period for the dispute process. Any party can call $\mathsf{AC.lock}$ with the list of signatures $\varSigma_{\mathcal{P}}$, $\Delta_{\mathsf{dispute}}$ and lockno to turn on the state channel. The application contract AC verifies all signatures and that lockno represents the largest counter received so far. If satisfied, AC sets $\mathsf{instantiated} := \mathsf{YES}$ and this disables all functionality within the application. Next AC creates the state channel contract SC which sets the list of participants $\mathcal{P}_1, ..., \mathcal{P}_n$ and the dispute timer $\Delta_{\mathsf{dispute}}$. Finally AC stores the state channel address SC.

Unlock application contract After the dispute process has concluded in SC, one party must send $\mathsf{state}'_\mathsf{i}, \mathsf{r}'_\mathsf{i}$ using $\mathsf{AC.unlock}$ before the functionality can be re-enabled. The application contract verifies that $\mathsf{state}'_\mathsf{i}$ indeed represents the final state by computing $\mathsf{hstate}'_\mathsf{i} := \mathsf{H}(\mathsf{state}'_\mathsf{i}, \mathsf{r}'_\mathsf{i})$, fetching the final state hash $\mathsf{hstate_i}$ from SC using $\mathsf{SC.getstatehash}$ and checking $\mathsf{hstate}'_\mathsf{i} = \mathsf{hstate_i}$. If satisfied, AC stores $\mathsf{state}'_\mathsf{i}$ and re-enables all functionality by setting $\mathsf{instantiated} := \mathsf{NO}$. Of course, if there is no activity within the state channel, then the state channel contract's dispute process can expiry without a submitted $\mathsf{hstate_i}$. In this case, the application contract verifies the state channel returns $\emptyset$ and re-enables all functionality without modifying the existing state.

4 Applying the Application Template for Battleship

We explore how to apply the application template from Section 3.3 to a contract like battleship (in Appendix A.2) such that it can be deployed within a state channel. Next we discuss workarounds (and pitfalls) discovered while building our proof of concept.

4.1 Minimal modifications for a State Channel

We present how to modify the battleship contract before deployment in order to support state channels. This tracks whether a state channel was instantiated, the lock/unlock functionality to instantiate the state channel, a new pre-condition for every function in the game and how to handle functionality with side-effects in the off-chain contract.

Applying the application template The application contract stores the dispute timer and a counter $\mathsf{instance}$ to track the number of times the state channel is turned on. It sets $\mathsf{instantiated} := \mathsf{NO}$ and both players $\mathcal{P}_1, \mathcal{P}_2$ for use by the state channel. The pre-condition **discard if** $\mathsf{instantiated} = \mathsf{YES}$ is included in every function except $\mathsf{BS.unlock}$. If the pre-condition is satisfied, then all future transactions that interact with this function will fail. This disables all functionality within the application contract if it is locked and the state channel is turned on.

Lock and unlock functions The lock function $\mathsf{BS.lock}$ requires a signature from both parties $\mathcal{P}_1, \mathcal{P}_2$ to authorise creating the state channel which is denoted as $\sigma_{\mathcal{P}}^{lock} := \mathsf{Sign}_{\mathcal{P}}('\mathsf{lock}', \mathsf{chan}_{\mathsf{ctr}}, \mathsf{round}, \mathsf{BS})$. Once the state channel is turned on, the battleship contract sets $\mathsf{instantiated} := \mathsf{YES}$, it creates a new state channel contract SC with the list of participants $\mathcal{P}_1, \mathcal{P}_2$ and the dispute timer $\Delta_{\mathsf{dispute}}$. The unlock function $\mathsf{BS.unlock}$ allows any party to submit the final game $\mathsf{state}_{\mathsf{i}}$ alongside the nonce r after the dispute proces is resolved in the state channel contract. The battleship contract verifies if it corresponds to the final state hash accepted by the state channel contract using $H(\mathsf{state}, r) == \mathsf{SC.getstatehash}$. If successful, the full state is stored and the flag $\mathsf{instantiated}$ is set as NO. This re-enables all functionality in the battleship contract.

4.2 Workarounds for State Channel

Off-chain contract Our proof of concept requires each player to deploy an off-chain version of the battleship contract to a local blockchain to replicate (and verify) the execution of all state transitions. Without modifying the local blockchain instance, both the off-chain and on-chain battleship contracts have different addresses. This poses problems for our fraud proofs if a message is signed for the off-chain contract address as it will not be valid when the on-chain contract is re-activated. To alleviate this issue, we sign two messages for the on-chain and off-chain contract. However there is an upcoming new consensus rule [3] to deterministically derive the contract's address which simplifies deploying an off-chain contract with the same address.

Loss of a global clock Both parties no longer share a global clock within the channel to self-enforce time-based events. We propose two approaches to handle time-dependent events. First, the time $\mathsf{t}_{\mathsf{challenge}}$ can be set by the player proposing a new state and the counterparty must verify the proposed time is within a range (i.e. a few minutes, or n blocks) before mutually authorising it. It must take into account the time required to turn off the channel via the dispute process and the time to initiate/settle the dispute such that $\mathsf{t}_{\mathsf{challenge}} := \mathsf{t}_{\mathsf{now}} + \Delta_{\mathsf{challenge}} + \Delta_{\mathsf{dispute}} + \Delta_{\mathsf{extra}}$. An alternative approach is to set $\mathsf{t}_{\mathsf{challenge}}$ as $\perp$ for all updates within the state channel. Instead the time $\mathsf{t}_{\mathsf{challenge}}$ is set by battleship contract when it is re-activated in the blockchain using $\mathsf{BS.unlock}$ and if the game is in a relevant phase.

No external interaction or side-effects We define a side-effect as a state update that relies on an environmental variable or external interaction with another contract. This is because the side-effects will not persist when the application is re-activated on the blockchain. Some examples in Ethereum include the environment variables $\mathsf{msg}, \mathsf{block}, \mathsf{tx}$, and transfering coins to another contract. All functions with side-effects should be deleted or disabled in the off-chain contract which for battleship includes the auxillery functions $\mathsf{BS.deposit}$ and $\mathsf{BS.withdraw}$.

Authenticating transaction signer and replay protection The battleship contract relies on $\mathsf{msg.sender}$ to authenticate the immediate caller as the transaction signer. This requires the party to sign a transaction for execution in the counterparty's local blockchain. Ethereum transactions have a $\mathsf{chain\_id}$ to prevent transactions signed for one blockchain being replayed to another blockchain. The counterparty can verify the transaction has set $\mathsf{chain\_id}$ and it is destined for the off-chain contract address before executing it in their local blockchain. Finally the off-chain contract can also include a new $\mathsf{BS.getstate}$ to return the full state and the corresponding $\mathsf{hstate}, \mathsf{i}$.

Persistent race conditions The gameplay for battleship is turn-based and it is clear which player is responsible for proposing every new state. Setting up the game using $\mathsf{BS.select}$ or $\mathsf{BS.begingame}$ has no order and both players may concurrently propose a state transition for the same version. In our case, both players can use a deterministic rule to resolve the race condition (i.e. $\mathcal{P}_1$ proposed state has priority) as the order of execution has no impact on the game's outcome. This highlights that race conditions in the underlying application are reflected in the state channel and can result in the state channel being turned off if the order of execution has an impact on the application's outcome.

Limitations due to the EVM The mapping data structure in Solidity for the Ethereum contract environment poses problems for the state channel as it cannot simply delete all key-value pairs. If a key-value pair is set to $\perp$ within the state channel, then this over-write must also occur when the full state is sent to the contract. Otherwise, the key-value pair will persist in the application contract after the state channel is turned off. For example, if a party's balance is set to

$\perp$ off-chain, but this isn't reflected in the on-chain contract, then this party can withdraw more coins than they deserve.

5 Proof of Concept Implementation

Step	**Purpose**	**Gas Cost**	**$$**
	Battleship Game		
1	Create BattleshipCon without State Channel	10,020,170	7.97
2	Deposit (BS.deposit)	44,247	0.04
3	Place bet (BS.placebet)	34,687	0.03
4	Select counterparty's ships (BS.select)	422,894	0.34
5a	Ready to play (BS.begingame)	47,651	0.04
5b	Do not play (BS.quitgame)	388,805	0.31
6	Attack (BS.attackcell)	69,260	0.06
7a	Reveal cell (BS.opencell)	73,252	0.06
7b	Reveal ship (BS.sunk)	111,372	0.09
8	Open ships (BS.openships)	159,748	0.13
9	Finish game (BS.finish)	275,521	0.22
10	Withdraw (BS.withdraw)	36,674	0.03
11	Fraud: Ships at same cell (BS.celltwoships)	280,766	0.22
12	Fraud: Declared not hit (BS.declarednothit)	284,261	0.23
13	Fraud: Declared not miss (BS.declarednothit)	284,654	0.23
14	Fraud: Declared not sunk (BS.declarednotsunk)	312,481	0.25
15	Fraud: Attack same cell (BS.attacksamecell)	100,861	0.08
16	Challenge period expired (BS.expiredchallenge)	75,349	0.06
	State Channel		
17	Create BattleshipCon with State Channel	13,607,0695	10.83
18	Lock (BS.lock)	991,617	0.79
19	Trigger dispute (SC.trigger)	84,106	0.07
20	Set state hash (SC.setstatehash)	70,035	0.06
21	Resolve (SC.resolve)	89,745	0.07
21	Co-operative turnoff (SC.close)	90,354	0.07
22a	Unlock (BS.unlock)	725,508	0.6
22b	Unlock (No Activity) (BS.unlock)	51,454	0.04
	Aggregated Statistics		
Turn state channel on and off		1,961,011	1.56
Average case for game		20,451,633	16.27
Worst case for game		30,237,372	24.05

Table 1: Costs of running the battleship game within the state channel. We have approximated the cost in USD ($) using the conversion rate of 1 ether = $306 and the gas price of 2.6 Gwei which are the real world costs in September 2018.

Purpose	**Propose**	**Verify**	**Acknowledge**
Place bet (BS.placebet)	232.18	212.23	0.44
Select counterparty's ships (BS.select)	330.59	304.70	0.44
Ready to play (BS.begingame)	243.70	224.51	0.44
Attack (BS.attackcell)	267.09	243.69	0.35
Reveal cell (BS.opencell)	268.93	248.51	0.40
Reveal ship (BS.sunk)	291.25	276.97	0.38
Open ships (BS.openships)	288.75	258.70	0.35
Finish game (BS.finish)	376.05	349.20	0.30

Table 2: Time taken to propose, verify and acknowledge new state transitions, measured in milliseconds (ms) and calculated as an average over 100 runs.

We present a proof of concept implementation for our battleship game within a state channel[5]. The experiment was performed using a Dell XPS 13 with Intel Core i5-7200U CPU @ 2.50GHz processor and 8GB LPDDR3 on a private Ethereum node using Ganache. In the following we discuss Table 1 which outlines the gas costs for our proposed modifications and Table 2 which presents a timing analysis to propose, verify and acknowledge a state transition within the channel.

Our experiment involves three contracts which includes the unmodified battleship contract (Step 1), the battleship contract after applying the application template (Step 15) and the state channel contract (Step 16). Deploying both the modified and unmodified battleship contract highlights the cost for modifying an application contract to support a state channel is approximately 1 million gas. A single game of battleship (Steps 4-9) via the blockchain costs \$16.27 (approx 20 million gas) where each player takes 65 shots[6]. In the worst case, the game requires one player to take 99 shots, and the counterparty to take 100 shots. This worst-case costs \$24.05 (approx 30 million gas) to finish the game. Locking the battleship game, creating the state channel, performing the dispute process costs and unlocking the battleship game costs \$1.56 (approx 1 million gas). The cost for each fraud proof is presented in Steps 11-14 and only one fraud proof is required per game to prove the counterparty has cheated.

All timings in Table 2 are approximations. We focus on the time taken to propose a new state transition, the time required for the counterparty to verify a state transition and for the initial proposer to verify the signed new state which is an acknowledgement from the counterparty that the state transition is complete. Proposing a new state takes between 232-376ms. This includes creating and signing a transaction at 12ms, executing the transaction within a local blockchain which is between 35-179ms (i.e. it depends on the function executed), retrieving the full new state from the local blockchain at 172ms, preparing a transaction for the counterparty and signing the full state's hash at 15ms. The state hash and signature is sent to the counterparty which incurs typical network latency. The counterparty takes between 212-349ms to verify a state transition which includes

[5] Anonymous code: https://www.dropbox.com/s/o5s5k662h9lqlk4/Battleship.zip?dl=0
[6] This number of shots is based on the better than random algorithm in. [7]

verifying the received transaction's signature (and checking it is destined for the off-chain contract) at 8 ms, executing the received transaction within the local blockchain which is between 34-163ms, retrieving the full new state from the local blockchain at 171ms, verifying the signature for the received state hash and verifying it matches the newly computed state hash at 0.4ms, and finally signing the new state hash at 4ms. The counterparty sends the corresponding signature for the new state hash back to the proposer which incurs typical network latency. Finally the proposer must verify the signature from the counterparty which takes 0.4ms. Overall, while the timings are reasonable for real-world use, the most expensive operations involve interacting with the Ganache client.

6 Discussion and Future Work

Supporting third party watching services To alleviate the security assumption that all parties must remain online and synchronised with the blockchain to watch for disputes, PISA [21] proposed that parties can hire an accountable third party to watch the channel on their behalf. The application-agnostic design of the new state channel construction Kitsune is beneficial to PISA as the accountable third party is only required to verify the state channel contract's bytecode (and not the application) before accepting a job from the customer. The accountable third party only requires a signature from every party in the channel $\Sigma_{\mathcal{P}}$, the state hash hstate and the version i to resolve disputes on the customer's behalf.

Funfair dilemma There is a chicken-and-egg problem on whether state channels should create and destroy applications off-chain, or if the state channel should first require an application to already exist on the blockchain. Perun and Counterfactual advocate for the former to minimise the up front cost of creating the channel, whereas Funfair are pursing the latter to minimise cost of resolving a dispute as only the application's state is kept off-chain. Fundamentally both approaches have a different trust assumption on the likelihood one party will trigger a dispute and whether the financial cost to resolve a dispute can interfere with the application. This dilemma can be summed up in a single question:

If the player is about to win a $10 bet, but the counterparty has stopped responding in the channel, then is it worthwhile for the player to turn off the channel, complete the dispute process, re-activate the application and win the bet via the blockchain if this process costs $100?

To evaluate this dilemma, our case study highlights that it costs $1.56 to resolve the dispute and submit the full game state to the contract which is an affordable (and reasonable) cost. However it does not consider the cost to deploy and instantiate the battleship game at $7.97, the continued cost for both players to play battleship or the remaining time required to finish playing it.

Dominant strategy to force-close Let's consider the worst-case for battleship. Both players set up the game with an expectation to play it within the state channel, but afterwards one player triggers a dispute to turn off the channel and the game must be finished via the blockchain. To play the entire game costs between $16.27 to $24.05 and every move requires a reasonable time period for moves to be accepted into the blockchain. If it is set to 5 minutes per move and the game requires 200 transactions, then the game may take several hours (i.e. 16 hours) to complete. This can be considered a dominant strategy by an adversarial player as it is likely rational players will simply forfeit their deposit (and bet) to quit the game early.

Inducing cooperative behaviour There is no mechanism to distinguish why a channel broke down, i.e. a blockchain cannot distinguish if Alice refused to sign and send Bob the latest state, or if Bob claims that he did not received a signed update. This makes it non-trivial to build a reputation system as it is unclear which party was at fault for the channel's failure and if any reasonable action can be taken to penalise the party at fault.To workaround the inability to identify the misbehaving party, future work must focus on how to induce cooperative behaviour amongst all parties in the channel. Any mechanism should not let an adversarial player to force-close a channel to their advantage (i.e. expecting rational players to simply give up). On the other hand, it must be careful not to discourage honest parties from closing the channel and continuing the application's execution via the blockchain.

Self-inspection of blockchain congestion On 6th January 2018, we witnessed the network's transaction fee spike to 95,788,574,583 wei [12][7] as the network became congested due to a significant increase in transaction throughput. Congestion impacts state channels as it increases the cost for resolving disputes (i.e. $57.58 for battleship) and continuing the application's execution (between $599 and $886 for battleship). If the increased transaction fees are not paid, then it is probable that a transaction will not be accepted into the blockchain within the dispute time period. Future work should focus on a new operation code (i.e. CheckCongestion()) that can retrospectively self-inspect the previous k of n blocks to determine if it was affordable for an honest party's transaction to be accepted into the blockchain. This could be used to extend the time period for resolving disputes and let players wait until the network is no longer congested before continuing the application's execution.

What to consider before deploying a state channel State channels require unanimous consent for an application's execution to progress off-chain. This implies an all parties should be involved throughout the entire application's execution or permit parties to leave via the blockchain without closing the channel. The developer must take care to ensure the application can gracefully handle (or remove) all race conditions. As well, they must be mindful the off-chain state size does

[7] The congestion was caused by a popular game called Cryptokitties.

not grow significantly which may prevent its publication to the blockchain. The application should be self-contained, not rely on any side-effects, and explicitly consider how to handle time-based events. Finally to guarantee liveness, it must always be reasonable to continue an application's execution via the blockchain.

Applicable Applications Our case study demonstrates that applications like battleship are not suitable for state channels due to the liveness requirement. Instead it appears that state channels are only useful for applications that are already suitable for execution via the blockchain and it only involves a small number of parties who can remain online throughout the application's life-time. It is also beneficial if all parties want to repeat the application's execution more than once such that the additional overhead to set up the channel costs less than simply executing it via the blockchain. Some potential applications include payments, casino games, boardroom elections and auctions. We conclude that a state channel should be viewed as an optimistic scaling approach only if all parties are willing to cooperate.

7 Acknowledgements

Patrick McCorry and Chris Buckland are supported by an Ethereum Foundation scaling grant, Ethereum Community Fund grant and a Research Institute grant. Andrew miller is supported by an NSF Grant 1801321. We thank the IC3-ETH participants Frank Sauer, Matthew Salazar, Oded Naor, and Deepak Maram for their support at kick-starting this project (and winning third prize). As well, we thank Tom Close for discussions around how to mitigate disputes in state channels. Finally we thank Master Workshop: Off the Chain for bringing together state channel researchers which helped bootstrap this paper.

References

1. Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.
2. Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. 2014.
3. Vitalik Buterin. Eip 1014: Skinny create2. Accessed 08/09/2018, `https://eips.ethereum.org/EIPS/eip-1014`.
4. Tom Close and Andrew Stewart. Force move games. Accessed 08/09/2018, `https://magmo.com/force-move-games.pdf`.
5. Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels, 2018.
6. Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

7. DataGenetics. Battleship. Accessed 08/09/2018, `http://www.datagenetics.com/blog/december32011/`.
8. Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.
9. Johnny Dilley, Andrew Poelstra, Jonathan Wilkins, Marta Piekarska, Ben Gorlick, and Mark Friedenbach. Strong federations: An interoperable blockchain solution to centralized third-party risks. *arXiv preprint arXiv:1612.05491*, 2016.
10. Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment channels over cryptographic currencies. Technical report, IACR Cryptology ePrint Archive, 2017: 635, 2017.
11. Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. Cryptology ePrint Archive, Report 2018/320, 2018. `https://eprint.iacr.org/2018/320`.
12. Etherscan. Ethereum gas price. Accessed 08/09/2018, `https://etherscan.io/chart/gasprice`.
13. Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.
14. Ethereum Foundation. Ethereum foundation grants update - wave iii. Accessed 08/09/2018, `https://blog.ethereum.org/2018/08/17/ethereum-foundation-grants-update-wave-3/`.
15. Ethereum Community Fund. Meet the grantees ecf class of 2018 part ii. Accessed 08/09/2018, `https://medium.com/ecf-review/meet-the-grantees-ecf-class-of-2018-part-ii-ff46a284a0b1`.
16. Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan 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.
17. Rami Khalil and Arthur Gervais. Nocust–a non-custodial 2 nd-layer financial intermediary.
18. Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.
19. ScaleSphere Foundation Ltd. Celer network: Bring internet scale to every blockchain. Accessed 08/09/2018, `https://www.celer.network/doc/CelerNetwork-Whitepaper.pdf`.
20. Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.
21. Patrick McCorry, Surya Bakshi, Iddo Bentov, Andrew Miller, and Sarah Meiklejohn. Pisa: Arbitration outsourcing for state channels. *IACR Cryptology ePrint Archive*, 2018:582, 2018.
22. Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*, 2017.
23. Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.
24. Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *Draft version 0.5*, 9:14, 2016.

25. Raiden. Raiden network. Accessed 08/09/2018, `https://github.com/raiden-network/raiden-contracts/blob/d3c30e6d081ac3ed8fbf3f16381889baa3963ea7/raiden_contracts/contracts/TokenNetwork.sol`.
26. Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159, 2016.
27. Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.
28. Jeremy Spilman. [bitcoin-development] anti dos for tx replacement. Accessed 08/09/2018, `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html`.

A Battleship Contract

We provide a high-level overview of the game battleship before proposing how to implement it as a smart contract. A security analysis for the game is included in Appendix C. We present how to convert the battleship game to support state channels using the template in Section 3.3.

A.1 Overview of Battleship

Battleship is a two-player game where each player has a list of ships that are placed on a 10x10 private board. Each ship must be marked in a straight line either horizontally or vertically. Our protocol only relies on a commitment to every player's ship and the signed messages exchanged between both parties in order to minimise long-term storage (and the associated gas-cost). An extension to this game is presented in Appendix B which includes a commitment for every cell on the board.

To set up the game, both parties exchange a commitment to their list of ships and the counterparty must submit it using BS.select. Afterwards both players can signal to begin the game using BS.begingame, otherwise they can quit using BS.gameover. In the turn-based gameplay, the player selects a cell to shoot using BS.attackcell and the counterparty must open the cell within a fixed challenge period. To open, the counterparty reveals if the cell is occupied by water or a ship piece using BS.opencell. If this shot sinks a full ship, then the counterparty must reveal the full ship (i.e. instead of the cell's opening) using BS.sunk. At the end, the winner must reveal their board and every ship's location to the loser using BS.openships. The loser has a fixed challenge period to prove if the winner's board was incorrectly set up or if the winner cheated during the game using a proof of fraud. A player can call BS.gameover after the challenge period has expired to finish the game.

A.2 Battleship Contract

We present each phase of the game, how to establish the contract, the turn-based gameplay and finally how the loser is provided an opportunity to prove the winner cheated.

Game Phases There are six phases $\mathsf{SETUP}, \mathsf{ATTACK}, \mathsf{REVEAL}, \mathsf{WIN}, \mathsf{FRAUD}, \mathsf{GAMEOVER}$. The SETUP phase is responsible for ensuring both players select a single list of ships to begin the game. Game play transitions between ATTACK and REVEAL as both players take a turn at shooting the counterparty's ships. The game transitions to WIN when one player wins the game and it will transition to FRAUD once the winner has opened all ship locations. This provides the loser a fixed time period to submit a proof of fraud that the winner's board is not well-formed or that the winner did not honestly reveal a cell during the game. Otherwise, the contract transitions to $\mathsf{GAMEOVER}$ and the winner can claim their winnings.

Contract establishment The contract is established with the address of both players $\mathcal{P}_1, \mathcal{P}_2$ and the challenge timer $\Delta_{\mathsf{challenge}}$. Both parties can deposit coins during SETUP phase before placing their bets.

Prepare list of ships A ship hash is denoted as $\mathsf{hship} := \mathsf{H}(x, y, x', y', r, \mathsf{round}, \mathcal{P}, \mathsf{AC})$ where x, y represents its starting co-ordinate, x', y' represents its finishing co-ordinate. Each party $\mathcal{P}$ computes and signs a list of ships:

$$\Sigma_1^N := \mathsf{Sign}_{\mathcal{P}}(((k_1, \mathsf{hship}_1), ..., (k_n, \mathsf{hship}_n)), \mathcal{P}, \mathsf{round}, \mathsf{AC})$$

Each ship in the list is denoted as (k, hship), where k is the length for that particular ship. This is sent to the counterparty who must submit it using $\mathsf{BS.select}$ and reserve the ships for the game.[8] Both players can notify the contract to begin the game using $\mathsf{BS.begingame}$ or one party can signal their desire to quit using $\mathsf{BS.gameover}$. Finally the game round is incremented regardless if it continues or not.

Game-play The contract maintains a counter move which is incremented after each player's turn. In the ATTACK phase, the player $\mathcal{P}$ challenges the counterparty to open a cell x, y by signing:

$$\sigma_{\mathcal{P}}^{shot} := \mathsf{Sign}_{\mathcal{P}}(x, y, \mathsf{move}, \mathsf{round}, \mathsf{AC})$$

This message is submitted using $\mathsf{BS.attackcell}$. It transitions the game phase to REVEAL and sets a fixed challenge period $\mathsf{t}_{\mathsf{challenge}} := \mathsf{t}_{\mathsf{now}} + \Delta_{\mathsf{challenge}}$ for the counterparty's response. The counterparty signs one of two messages depending on whether a ship was sunk:

[8] In Appendix C.1 we present a cut-and-choose protocol to allow the counterparty probabilistic verify the board is well-formed.

$$\sigma_{\mathcal{P}}^{hit} := \mathsf{Sign}_{\mathcal{P}}(x, y, b, \mathsf{move}, \mathsf{round}, \mathsf{AC})$$
$$\sigma_{\mathcal{P}}^{sunk} := \mathsf{Sign}_{\mathcal{P}}(x, y, x', y', r, \mathsf{hship}, \mathsf{move}, \mathsf{round}, \mathsf{AC})$$

The counterparty is responsible for submitting either signed message. The first message declares if the cell is marked with water ($b = 0$) or a ship location ($b = 1$). It is submitted using BS.opencell. The second message declares the shot sank a ship and requires the counterparty to open the corresponding ship commitment hship to BS.sunk. Each party must keep a copy of every signed message[9] as it can later be used to prove fraud which we discuss in Section A.4. The game transitions to WIN if one player has declared all their ships sunk.

End of game After one player has lost the game (or if the contract has detected cheating by the loser as illustrated in Section A.3), the winner must open their remaining ship commitments using BS.openships. This contract transitions to FRAUD which provides a fixed challenge period for the loser to submit a proof of fraud. After this time period, the winner can redeem their reward using BS.gameover and the game transitions to GAMEOVER. Of course, if both parties have cheated, then the winnings are simply burnt.

A.3 Checking for Fraud

We present integrity checks the contract can perform throughout the game to verify that either party has not cheated. These checks are performed whenever a player calls BS.attackcell, BS.sunk, BS.opencell and BS.openships.

Exceeded maximum number of moves The contract maintains three counters. The first move keeps track of the number of actions taken by bother players. If move exceeds the number of possible moves in the game for both players, then the contract can confirm that both players have cheated as an honest player will have declared all their ships as sunk before the limit for move is exceeded. In this case, both players are set as cheating and the game transitions to GAMEOVER without a winner. Both hits_i and water_i keeps track of each player's attack on the counterparty's board. If hits_i exceeds the number of ship positions on the board or $\mathsf{water_i}$ exceeds the possible number of water cells, then the counterparty was dishonest about their cell opening. In this case, the counterparty is marked as cheated, the game transitions to WIN and the winner must open their ships.

Players only play using valid cells All cells must be within the permitted range $0 <= x < 10$ and $0 <= y < 10$ for any signed message received.

A ship was not placed horizontally or vertically The contract can check whether an opened ship was placed on the board horizontally or vertically. To verify, it checks that every location for a ship either has the same x or y co-ordinate, and that x or y is incremented (or decremented) strictly by one for every ship location. It also checks the ship's length which is established during set up.

[9] Every signed message is emitted by the contract and thus it is easily fetchable.

A.4 Proof of Fraud

To alleviate the need to validate the entire game within the smart contract environment (and incurring unreasonable gas costs), the protocol is designed to let each player validate the game and submit a proof of fraud if the counterparty has cheated. In the following we present the fraud proofs that can be verified by the contract.

Player has shot the same cell twice The contract cannot independently verify if a player has shot the same cell twice as it does not store the opening of cells. Instead the counterparty can submit the two signed shots $\sigma_{\mathcal{P}}^{shot}, \sigma_{\mathcal{P}}^{shot'}$, the corresponding $\mathsf{move}, \mathsf{move}'$ counters and the cell x, y using $\mathsf{BS.attacksamecell}$. The contract verifies if the signatures are valid (and from the same party), both shots are for the same cell, and $\mathsf{move} \neq \mathsf{move}'$. This proof of fraud can be submitted to the contract at any point during the game.

Counterparty was dishonest about a cell opening The counterparty has marked a cell (x, y) as water, but an opened hship states it is a ship location. To prove fraud, the player submits the ship identifier hship, the disputed cell x, y and the signed opening of the cell $\sigma_{\mathcal{P}}^{hit}$ using $\mathsf{BS.declarednothit}$. The contract can verify if this cell opening was signed by the counterparty as $b = 0$ and the ship hship claims to be at x, y. On the other hand, the counterparty may also mark a cell as a ship location, but no ships are at that location. This proof of fraud is similar as the player submits the disputed cell location x, y alongside its signed opening $\sigma_{\mathcal{P}}^{hit}$ using $\mathsf{BS.declarednotwater}$. The contract is satisified if it cannot find a ship at that location. Both proofs can only be submitted during FRAUD.

Two ships claim to be at the same cell The cheater has used the same cell for two or more ships. The index for both ships and the cell x, y must be submitted to the contract using $\mathsf{BS.celltwoships}$. The contract looks up the co-ordinates for each ship and checks if it claims to be at the same location x, y. This proof is applicable during FRAUD after all ships are opened by the winner.

Ship was not declared as sunk The counterparty did not declare a ship as sunk. All signed cell openings $\sigma_{\mathcal{P},1}^{hit}, ..., \sigma_{\mathcal{P},k}^{hit}$ and the ship identifier hship must be submitted to the contract using $\mathsf{BS.declarednotsunk}$. This allows the contract to verify that every ship location was opened and this implies the counterparty did not declare the ship as sunk as the final opening should be $\sigma_{\mathcal{P}}^{sunk}$. This proof is applicable during FRAUD after all ships are opened by the winner.

Challenge period has expired The contract relies on a global clock (i.e. block timestamp or block height) for the challenge period $\Delta_{\mathsf{challenge}}$. If a player does not respond within this time period, then the counterparty can notify the contract using $\mathsf{BS.expiredchallenge}$ and the counterparty is set as the winner if the challenge period has expired.

B Full Board Extension

We present an extension to our battleship game which requires a commitment for every cell of the board in addition to the ship commitments. For each cell, the commitment consists of a flag b indicating if it is occupied by water $b := 0$ or a ship $b := 1$, and a randon nonce r. Assuming the selected board is well-formed, then it can prevent each player lying about their cell opening during the game, but it also increases the game state and the gas requirement for each move in the game.

B.1 Modifications to the Battleship Contract

We present how to modify the battleship contract to support the full board extension. This requires modifying how the game is prepared, how a cell opening during the game play is verified by the contract and how the full board is opened at the game's end.

Prepare boards Our extension requires each party to compute an entire board to accompany a list of ships. The board is a list of cell hashes such that $\mathsf{hcell}_{1,1}$, ..., $\mathsf{hcell}_{n,n}$ where n, n is the final grid co-ordinate. A cell hash is $\mathsf{H}(b, r, \mathsf{round}, \mathcal{P}, \mathsf{AC})$, where b is a flag indicating if it is occupied by water $b := 0$ or a ship location $b := 1$, and r is the nonce. The party signs the list of ships and the board cells:

$$\sigma := \mathsf{Sign}_{\mathcal{P}}(((k_1, \mathsf{hship}_1), ..., (k_n, \mathsf{hship}_n)), (\mathsf{hcell}_1, ..., \mathsf{hcell}_n), \mathcal{P}, \mathsf{round}, \mathsf{AC})$$

The contract stores every cell hash in the contract for future use. Each party is responsible for reserving the list of ships and the board on behalf of their counterparty using $\mathsf{BS.select}$. All remaining $N-1$ list of ships and their corresponding boards must be opened and reviewed by the counterparty. If satisified, each party notifies the contract to begin the game using $\mathsf{BS.begingame}$ or they can quit using $\mathsf{BS.gameover}$.

Game-play Our extension requires modifying how a player responds to an attacked cell:

$$\sigma_{\mathcal{P}}^{hit} := \mathsf{Sign}_{\mathcal{P}}(x, y, b, r_{cell}, \mathsf{move}, \mathsf{round}, \mathsf{AC})$$
$$\sigma_{\mathcal{P}}^{sunk} := \mathsf{Sign}_{\mathcal{P}}(x, y, x', y', r_{cell}, r_{ship}, \mathsf{hship}, \mathsf{move}, \mathsf{round}, \mathsf{AC})$$

Both the hit and sunk messages include the nonce r_{cell}. This lets the contract open $\mathsf{hcell}_{x,y}$ and confirm that the supplied b matches the commitment during the setup. The opening can be stored by the contract, otherwise each party must keep a copy of every signed message[10] for future fraud proofs as presented in Section A.4.

[10] Every signed message is emitted by the contract and thus it is easily fetchable.

B.2 Changes to Fraud Detection

We present the additional fraud detection that is performed by the contract and the player due to the extension.

Cut-and-choose protocol To set up the game, both parties participate in a cut-and-choose protocol to provide a probabilistic guarantee that the counterparty's board is well-formed. Each player commits, signs and sends the counterparty N boards (i.e. list of ship and cell commitments). The counterparty reserves one board for the game using $\mathsf{BS.select}$. Once selected, each player reveals the remaining $N-1$ boards to the counterparty who verifies the boards are well-formed. If both parties are satisified, then they can signal to begin the game using $\mathsf{BS.begingame}$, otherwise they can quit using $\mathsf{BS.gameover}$. While this provides a probabilistic guarantee the board is correctly set up, it does not let each player place the ships on their board which may remove an element of the game.

Integrity checks As presented in Section A.3 the contract checks all signed messages received to self-enforce the game's correct execution. Our extension requires the contract to check every cell opening with the stored cell hash hcell.

Proof of fraud If we assume the board is well-formed upon set up, then the party cannot be dishonest about their cell opening during the game. The fraud proofs $\mathsf{BS.declarednothit}$ or $\mathsf{BS.declarednotwater}$ are still required as the board used in the game can be invalid and the contract must verify that the cell opening does not correspond to a ship opening. There is no change to the fraud proof except that the cell nonces are submitted to the contract alongside the signed cell openings.

C Security Analysis for Battleship Game

We provide a brief security analysis for the battleship game and demonstrate how the fraud proofs can be used to self-enforce the game's correct execution. This includes how the contract can detect if a board is not well-formed, how it self-enforces a player to attack a valid cell and how to ensure the corresponding cell is honestly opened. Finally we highlight the contract forfeits any payout if both players are caught cheating.

C.1 Detecting an Invalid Board

A cut-and-choose protocol lets each player select one of the counterparty's committed boards at random for use in the game and afterwards review the remaining $N-1$ boards before deciding to play the game. This provides a probabilistic guarantee the selected board is well-formed, but it is not a mandatory step the contract can self-enforce. Both players may decide to only send a single board commitment to each other so they can manually place the ships. This provides an opportunity for one (or both) players to construct an invalid board and we highlight how the contract can detect it.

Overlapping ships The board is invalid if one cell is used for more than one ship. The fraud proof $\mathsf{BS.celltwoships}$ can be used to prove that ships are overlapping, but it requires the ship openings to be revealed. There is no guarantee the counterparty will reveal both ship openings during the game, but the winner is always required to open all ships and thus the loser is always provided an opportunity to provide this fraud proof to the contract.

Ship is not horizontal or vertical All ships must be placed horizontally or vertically on the board, and it must be in a straight line. No fraud proof is required as the contract is responsible for checking every ship opening. We outline in Section A.3 how the contract checks that a ship was placed on a list of valid cells and how it can check if the ship is placed horizontally or vertically.

Placed ships are not the correct size The board is invalid if a ship does not occupy the correct number of cells on the board. The contract stores a list of sizes for each ship. Each ship is represented as (k, hship) and the contract checks that k corresponds to the expected size for the ship at this position in the list. When the opening of hship is revealed to the contract it will check the number of cells used by the ship corresponds to k.

Not placing a ship on the board The board is invalid if a ship is not placed on the board. The contract requries a commitment hship for every ship before the game can begin. If the commitment's pre-image is not well-formed (i.e. it is $\perp$ or the ships location is not occupying valid cells), then the contract will not accept the ship opening. Thus after the challenge period $\mathsf{t_{challenge}}$, the contract will assume the player has not responded with a ship opening. On the other hand, if the ship's location is not well-formed then the fraud proofs highlighted above can be used.

Placing extra ships on the board The contract only accepts a fixed number of ship commitments and thus the contract self-enforces that only the correct number of ships are placed on the board.

C.2 Attacker during Game Play

The contract self-enforces the turn-based game play and whose turn it is to attack. We consider how a cheater can manipulate the attack message $\sigma_{\mathcal{P}}^{shot}$ that is supplied to $\mathsf{BS.attackcell}$.

Preventing replay attacks The contract is responsible for tracking (and incrementing) two counters. The counter round is incremented for every new battleship game in this contract (incuding if the game set-up is restarted) and move is incremented for every new move within a single game. Both counters are used to prevent replay attacks from previous battleship game or moves within this game. All messages also include the battleship contract address BS.

Attacking an invalid cell The player must select a single cell to attack and as outlined in Section A.3 the battleship contract verifies the proposed cell is valid.

Attacking same cell twice In order to reduce storage, the battleship contract does not keep track of all prevously attacked cells. In Section A.4 we present how the counterparty can submit two signed attack messages $\sigma_{\mathcal{P}}^{shot}, \sigma_{\mathcal{P}}'^{shot}$ to the contract using $\mathsf{BS.attacksamecell}$ to demonstrate the party has tried to attack the same cell twice.

Not attacking any cell The player can abort and not attack any cell. After the challenge time $\mathsf{t}_{\mathsf{challenge}}$ has expired, the contract assumes the player has aborted and sets the counterparty as the winner.

C.3 Revealer during Game Play

After a cell is attacked, the contract requires the counterparty to open the cell with $\sigma_{\mathcal{P}}^{hit}$ or declare a ship as sunk with $\sigma_{\mathcal{P}}^{sunk}$.

Opening a different cell The battleship contract stores the co-ordinates x, y for the attacked cell and it will only accept a cell (or ship) opening if it is for the stored co-ordinate.

Dishonest about cell opening If the counterparty is not honest about the cell opening, then the fraud proofs outlined in Section A.3 (i.e. $\mathsf{BS.declarednothit}$ or $\mathsf{BS.declarednotwater}$) can be used after the cheater has won and revealed the opening of all their ships. This is comparable to playing the game in-person as the counterparty is not forced to reveal all ships until the game's end. We provide an extension in Appendix B that requires each party to provide a commitment for every cell on the board to prevents this issue (i.e. if the board is set up correctly), but it increases the cost to play the game. As well, we highlight in Section A.3 the contract keeps tracks on the number of moves played and the game will always finish when this limit is exceeded.

Not declaring a ship as sunk If the final ship location is hit, the counterparty can simply not declare the ship as sunk. Instead, the counterparty has to open the attacked cell as water or a ship location. No more cells for this ship can be hit and thus the ship cannot be opened during the game play. This requires the players to wait until the game has finished and the cheater to be set as the winner. The loser can provide a proof of fraud as presented in Section A.4 to prove the ship was never declared as sunk. We highlight the extension presented in Appendx B cannot prevent this issue as it is not straight-forward to distinguish several cell openings as being a single ship or several adjacent ships. While the cheater can never win the game, they can force the counterparty to play until the game's end.

Not opening any cell or ship The counterparty can decide not to open any cells (or ships) in response to an attacked cell. If there is no response by the challenge time $\mathsf{t}_{\mathsf{challenge}}$, then the contract will assume the counterparty is no longer responding and the counterparty is set as the winner.

C.4 Both players are cheating

The battleship contract should not issue any payout if it is discovered that both players have cheated. After the contract has detected cheating by one player, it always transitions to WIN and sets the counterparty as the winner. This requires the counterparty to open all ships and a fixed challenge period is provided for the cheater to submit a proof of fraud. If both players are caught as cheating, then the contract transitions to GAMEOVER and forfeits the payout.

```
Template for application contract

$\mathsf{instantiated} := \perp, \mathsf{state} := \perp$
$\mathcal{P} := \emptyset, \Delta_{\mathsf{dispute}} := 0,$
$\mathsf{SC} := \perp, \mathsf{lockno} := 0$

constructor $(\mathcal{P}')$:

    set $\mathcal{P} := \mathcal{P}'$
    set $\mathsf{instantiated} := \mathsf{NO}$

function example():

    discard if $\mathsf{instantiated} = \mathsf{YES}$
    _;

function lock($\Delta'_{\mathsf{dispute}}, \Sigma_{\mathcal{P}}$):

    discard if $\mathsf{instantiated} = \mathsf{YES}$
    if $\mathsf{VerifySig}(\mathcal{P}, (\text{“instantiate”}, \mathsf{AC}, \mathsf{lockno}), \Sigma_{\mathcal{P}})$
        set $\mathsf{instantiated} := \mathsf{YES}$
        set $\mathsf{lockno} := \mathsf{lockno} + 1$
        set $\mathsf{SC} := \mathsf{StateChannel}(\mathcal{P}, \Delta_{\mathsf{dispute}}, \mathsf{this})$

function unlock($\mathsf{state}', \mathsf{r}'$):

    discard if $\mathsf{instantiated} = \mathsf{NO}$
    if $\mathsf{H}(\mathsf{state}', \mathsf{r}') = \mathsf{SC.getstatehash}()$
        $\mathsf{instantiated} := \mathsf{NO}$
        $\mathsf{state} := \mathsf{state}'$
    else if $\perp = \mathsf{SC.getstatehash}()$
        $\mathsf{instantiated} := \mathsf{NO}$
```

Fig. 1: The application contract template. The above modifications must be included to support a state channel. It allows all functionality to be disabled when the channel is created and re-enables all functionality after the dispute process when provided with the full state.

```
State channel contract

$\mathsf{status} := \bot$
$\mathcal{P} := \emptyset, \mathsf{AC} := \bot,$
$\mathsf{hstate} := \bot, \mathsf{i} := 0$
$\Delta_{\mathsf{dispute}} := 0, \mathsf{t}_{\mathsf{now}} := 0, \mathsf{t}_{\mathsf{end}} := 0$

constructor $(\mathcal{P}', \Delta'_{\mathsf{dispute}}, \mathsf{AC}')$:

    set $\mathcal{P} := \mathcal{P}'$
    set $\Delta_{\mathsf{dispute}} := \Delta'_{\mathsf{dispute}}$
    set $\mathsf{AC} := \mathsf{AC}'$
    set $\mathsf{status} := \mathsf{ON}$

function triggerdispute$(\sigma_k)$:

    discard if $\mathsf{status} \neq \mathsf{ON}$
    discard if $\mathcal{P} \notin \mathcal{P}_k$
    if $\mathsf{VerifySig}(\mathcal{P}_k, (\mathsf{SC}, \mathsf{AC}, \text{“dispute”}), \sigma_k)$
        set $\mathsf{status} := \mathsf{DISPUTE}$
        set $\mathsf{t}_{\mathsf{start}} := \mathsf{t}_{\mathsf{now}}$
        set $\mathsf{t}_{\mathsf{now}} + \Delta_{\mathsf{dispute}} := \mathsf{t}_{\mathsf{start}} + \Delta_{\mathsf{dispute}}$

function setstatehash$(\mathsf{hstate}', \mathsf{i}', \Sigma_{\mathcal{P}})$:

    discard if $\mathsf{status} = \mathsf{OFF}$
    discard if $\mathsf{i}' \leq \mathsf{i}$
    if $\mathsf{VerifySig}(\mathcal{P}, (\mathsf{hstate}', \mathsf{i}', \mathsf{SC}, \mathsf{AC}), \Sigma_{\mathcal{P}})$
        set $\mathsf{hstate} := \mathsf{hstate}'$
        set $\mathsf{i} := \mathsf{i}'$

function resolve():

    discard if $\mathsf{status} \neq \mathsf{DISPUTE}$
    discard if $\mathsf{t}_{\mathsf{now}} < \mathsf{t}_{\mathsf{end}}$
    set $\mathsf{status} := \mathsf{OFF}$

function getstatehash():

    discard if $\mathsf{status} \neq \mathsf{OFF}$
    return $\mathsf{hstate}_{\mathsf{i}}$

function getdispute():

    discard if $\mathsf{status} \neq \mathsf{OFF}$
    return $(\mathsf{t}_{\mathsf{now}}, \mathsf{t}_{\mathsf{end}}, \mathsf{i})$
```

Fig. 2: The state channel contract for Kitsune. It is responsible for managing the dispute process and determining the final state hash. Discard fails the transaction execution if the pre-condition is satisfied.