Game Channels: state channels for the gambling industry with built-in PRNG

Alisa Cherniaeva[1], Ilia Shirobokov[1], and Alexander Davydov[2]

[1]BlockProof Tech LLC*
[2]DAO.Casino†

November 2018‡

Abstract

Blockchain technology has immense potential. At the same time, it is not always possible to scale blockchains. State Channels solve the problem of scalability while increasing the blockchain's speed and efficiency. State Channels present a workaround to current blockchains' TPS (transaction per second) bottleneck. We used State Channels as a foundation and created Game Channels. We built it around the needs of the gambling market. We also developed Signidice PRNG as well as a dispute resolution mechanism. Signidice uses unique digital signatures and is also described below. The potential use of Game Channels technology is not only gambling; some types of online gaming may also be able to use it.

Contents

*research@blockproof.tech
†research@dao.group
‡Revised September 2019

1 Introduction

Video games in general, and gambling in particular, require fast interaction between players. Yet most blockchain systems fail to offer high–speed block generation, and the average transaction time is too long as well. For instance, Ethereum generates a new block approximately every 15 seconds [12], while each transaction on average takes 6 minutes [18]. Obviously, this is too slow for gaming. Moreover, the transaction fees required will increase the overall session costs.

Layer 2 solutions [20] are designed to remedy this situation. These operate "on top" of existing blockchains. One of these solutions is *state channels*. In this paper we will cover DAO.Casino's state channels, focusing on gambling implementation options.

1.1 State Channels overview

State channels incorporate instant zero–fee settlements between two channel parties introduce generalized extended functionality to the blockchains with which they are associated. A State Channel operates as follows [4]:

1. Part of the blockchain state is locked via a smart contract, so that a specific set of participants must completely agree with each other to update this state. This state is called the *state deposit*.

2. Participants update the state amongst themselves by constructing and signing transactions that are submitted to the blockchain.

3. Finally, participants store the state back to the blockchain, which closes the state channel.

In the ideal scenario blockchain participants interact with the channel twice: when the channel is opening and and when the channel is closed in an authorized manner with the updated state deposit.

Should participants disagree over a result, the channel state is then changed to a *dispute*; the smart contract then acts as an arbitrator in the *dispute resolution* process.

1.2 Requirements for state channels

Before going into any further detail about state channels, we need to define the basic security requirements. Coleman, Horne and Xuanji [5] specify the following two requirements:

- Trustlessness - parties who entrust state to a properly initialized state channel should not significantly increase the risk of that state being manipulated.

- Finality - state channel operations have the same degree of finality and irreversibility as the analogous operations performed directly on the chain itself.

Additional security and computational requirements come from the gambling industry:

- Minimal communication complexity in state channels.

- Provably fair random number generation support.

- Minimal time required for random number generation.

- Instant verification of randomly generated numbers.

- Once either party makes a bet, one of the parties has to pay the debt to the other at some point.

As long as the above requirements are met, fair and convenient gambling is available to all parties involved.

1.3 Related works

Payment channels [1] are similar to state channels, but a state deposit stores only participant balances. Payment channel networks are built from multiple separate channels that can be coupled when needed. The most popular implementations of the approach are The Lightning Network for Bitcoin [15] and The Raiden Network for Ethereum [16]. Given that payment channels store no game data, using them in the industry is not practical.

State channels were first described in detail by Jeff Coleman in [4] and were later considered in other works [21, 14, 8]. Generalized State Channels defined in [5] represent another step forward, enabling users to install new functionality into an existing channel without touching the blockchain. A similar idea was independently developed and implemented in Perun [6]. To make state channels more trustworthy, McCorry *et al* [13] defined tools preventing execution fork attacks. However, all these works considered general state channel applications without focusing on the needs of the gambling industry.

Daniel Kraft independently described protocol for turn-based games which also called Game Channels [9]. This protocol was implemented as part of XAYA Game Library [24]. FunFair [10] offers a proprietary implementation of state channels called Fate Channels. In particular, it includes a provably fair random numbers generator based on the commit–reveal pattern for seed generation. Magmo [3] developed a framework supporting multiplayer channels. Acebuster designed state channels tailored to poker [11]. It is noteworthy that all state channel implementations have limitations in terms of the game types supported.

1.4 DAO.Casino's solution

For the purposes of this paper all games meeting the following conditions shall be defined as simple pvp–games:

- Two parties only (e.g. a casino vs. a player);
- The game logic may require random number generation;
- When either party makes a move, a certain end state can be traced to choose the winner and/or distribute funds among the participants.

We offer a *Game Channel* technology that allows launching blockchain-based simple pvp–games without fees for additional transactions, and with zero delays between moves/rounds. This solution meets the above–mentioned requirements for state channels.

Game channels is an instance of state channels tailored to the needs of the gambling industry. Modification of the Signidice [19] algorithm is used for provably fair random number generation within a game channel. Signidice generates random numbers from a unique digital signature of one player and a random seed from the other player. It allows a reduction in communication complexity in comparison with the commit–reveal RNG scheme.

This paper is organized as follows. Section 2 gives basic notations, definitions and a simplified scheme for game channels. In section 3 we provide detailed coverage of the orignal game channel protocol. Section 4 covers a modification of the original protocol for the two player case and comments on delegating blockchain interactions to a third party.

2 Preliminaries

In this section we introduce definitions and notation that will be used throughout the paper.

We denote the set of integers modulo an integer n by $\mathbb{Z}_n$. When writing $x \xleftarrow{\mathrm{R}} S$ we mean that x is chosen uniformly at random from the set S. By H we denote some cryptographic hash function.

The communication model considers a point–to–point channel between two parties. One of them may be a malicious adversary. The adversary can diverge from the specified protocol in any way.

2.1 Safety and liveness of game channels

Liveness of game channels assures that there is always a result for each of the channel participants within a round. Round results within the channel can be saved on–chain any time, with a submission window of sufficient size left to avoid miner attacks. Thus, game channels rely on the blockchain liveness. Simply put, the exact state sent by the channel participants must always be locked in the blockchain. It is assumed that the state is accepted and signed by all channel participants. Given the above, it is important that all participants have stable Internet connections free of significant interruptions.

The safety of a game channel also relies on the safety of the blockchain. It assures that all game channel participants obtain a valid and identical result. Analyzing intra–channel activities, we can suggest a case when a participant does not respond to messages sent to him via the channel. There is no telling whether an actual network failure occurred or a participant deliberately gave no response. As far as state channels are concerned, the participant availability

issue is expected to be resolved through implementing some procedure for obtaining a valid result even when one of the participants is unavailable. In game channels the relevant functions are integrated into the smart contract responsible for dispute resolution; the smart contract is authorized to decide the reward distribution in compliance with the game logic. Note that a new round can only be started after all the parties have agreed upon the previous round. Thus, in the worst case one player will be in the $n + 1$ state, while the other is in the n state. If the first player fails to send a state update, the dispute is resolved in favor of the second player, who receives the maximum reward.

Griefing is the ability of a participant to deviate from the protocol in order to disrupt participant interaction without directly violating the security of the protocol. There are two griefing strategies. One implies forcing a party into paying the channel closure transaction; to do so, a party that was supposed to close the channel does not send the transaction to the blockchain. This is not a major issue, as state deposits considerably exceed transaction costs, and participants are likely to assume this risk. The other strategy implies posting expired channel states during a dispute if an attacking party believes that the other participant is unavailable. To reduce this risk, a game channel checks the game round number when there is a state change attempt in dispute; if the difference between the round number of the new state and the round number suggested by an attacker exceeds 1, the attacker loses their deposit.

It is noteworthy that an incorrect implementation of game channels and software errors can cause one or both players to lose their entire deposits. However, this issue is out of the scope of the present work.

2.2 Signatures and Fingerprints

Definition 1. A signature scheme Σ is a tuple $(M, S, K, KeyGen, Sign, Verify)$ where:

- M is a finite field of possible messages;
- S is a finite field of possible signatures;
- K is a finite field of possible keys;
- $KeyGen : (1^k) \rightarrow (sk, pk)$. This algorithm takes as input a security parameter k and outputs secret and public keys;
- $Sign_{sk} : (m) \rightarrow \sigma$. The signing algorithm takes as input a message $m \in M$ and secret key $sk \in K$, and outputs a signature $\sigma \in S$;

- $Verify_{pk} : (m, \sigma) \to \{0, 1\}$. This algorithm check whether the signature $\sigma \in S$ for a message $m \in M$ and a public key $pk \in K$ is valid.

In addition we define the uniqueness property for a signature scheme.

Definition 2. A signature scheme Σ is called *unique* if for every message $m \in M$ and for every public key $pk \in K$ there is only one valid signature $\sigma \in S$.

In our protocols we use two types of signature schemes: RSA [17] and ECDSA [7]. ECDSA is a standard signature for transaction acknowledgement within a channel. RSA is reserved for pseudorandom number generation in the Signidice algorithm.

Remark. RSA can be replaced by any other signature with the uniqueness property. We suggest considering BLS [2] as the primary alternative.

When a channel is open, the smart contract only stores the *Merkle–tree fingerprint* and not the entire RSA public key. The purpose of this is to reduce the transaction cost of opening a channel.

Definition 3. Let $pk = (N, e)$ be a RSA public key. Then $f = H(H(e), H(N))$ is a Merkle–tree fingerprint of the RSA public key pk.

Remark. In the DAO.Casino implementation $KECCAK - 256$ [23] is always used for the H function to ensure compatibility with Ethereum.

2.3 Signidice

Signidice [19] is a protocol that allows pseudorandom number generation by two parties.

Define the bit hash length as $hash.size$, and the maximum and the minimum number the generation can yield as max and min respectively.

Remark. If $max - min$ is a power of two, the while loop in step 2 of the Signidice algorithm can be omitted. But if it isn't a power of two, the resulting distribution is not uniform, as some numbers may be more likely than others.

In the DAO.Casino implementation the unique RSA unique signature is used by the $Sign$ and $Verify$ functions:

- $RSA.Sign_d(m) : \mathbb{Z}_n \to \mathbb{Z}_n : m \to m^d mod N$
- $RSA.Verify_e(m, s) : \mathbb{Z}_n \times \mathbb{Z}_n \to \{True, False\} : (m, s) \to$ check if $s^e mod N == m$

Signidice

1. *Alice* send $seed \xleftarrow{R} \{0,1\}^*$ to *Bob*
2. Bob computes:
   $h \leftarrow H(seed)$
   $S \leftarrow \Sigma.Sign_d(h)$
   $L \leftarrow H(S)$
   $range \leftarrow max - min + 1$

while $L \geq \lfloor (2^{hash.size} - 1)/range \rfloor \cdot range$ **do**
   $L \leftarrow H(L)$
end while
   $L \leftarrow (L \bmod range) + min$

3. *Bob* send S, L to *Alice*
4. Alice check results:

if $\Sigma.Verify_e(S)$ and L is correct **then**
   The number L is accepted
else
   The number L is not accepted
end if

2.4 Channels

As stated above, State Channel operation requires an on–chain smart contract. Let's define the core functionalities required to enable this contract:

- *OpenChannel* - Defines the parties' consent to interact within the channel;
- *UpdateChannel* - Changes the last state stored in blockchain to the newest state approved by both parties;
- *CloseChannel* - Completes the channel operation. The latest state approved by both channel parties is stored in the blockchain; funds are distributed according to the state;
- *OpenDispute* - The game is halted and a dispute is initiated when one of participants fails to get reliable relevant data. A dispute has two potential outcomes specified below;
- *ConsensusResolve* - Parties resolve the dispute by consenting a new state;

- *ArbitrationResolve* - The dispute is resolved through smart contract arbitrage.

Remark. Note that these functionalities may be implemented within a single contract or within several interacting contracts.

Now we introduce the following concept.

Definition 4. The set $(OpenChannel, UpdateChannel, CloseChannel, OpenDispute, ConsensusResolve, ArbitrationResolve)$ is called a *Game Channel Contract System* and is denoted by *GCCS*.

Remark. GCCS can also be extended by adding additional functionality, but these addons are out of scope in the present work.

Definition 5. We say that a connection between two parties is called a *Channel* if the following conditions holds:

1. Every message sent contains some game–related data;
2. Every message (or main part of a message) is signed by the sender;
3. The connection verified by GCCS.

Each channel has a *state*. The channel state is the last message sent by a participant that unambiguously defines the latest game state and/or the participants' balance. Note that each protocol participant must store the latest channel state.

Also, each channel has a $lifetime$ parameter. The $Lifetime$ defines the number of blockchain blocks available to the GCCS to update the channel state and open disputes related to this state.

We define the channel participants as: *Player* and *Dealer*. The table below defines the differences between these roles:

Player	**Dealer**
Makes bets	Receives bets
Generates PRNG seed	Generates pseudorandom numbers
Checks game results	Calculates game results

Other differences depend on the specific protocol implementation.

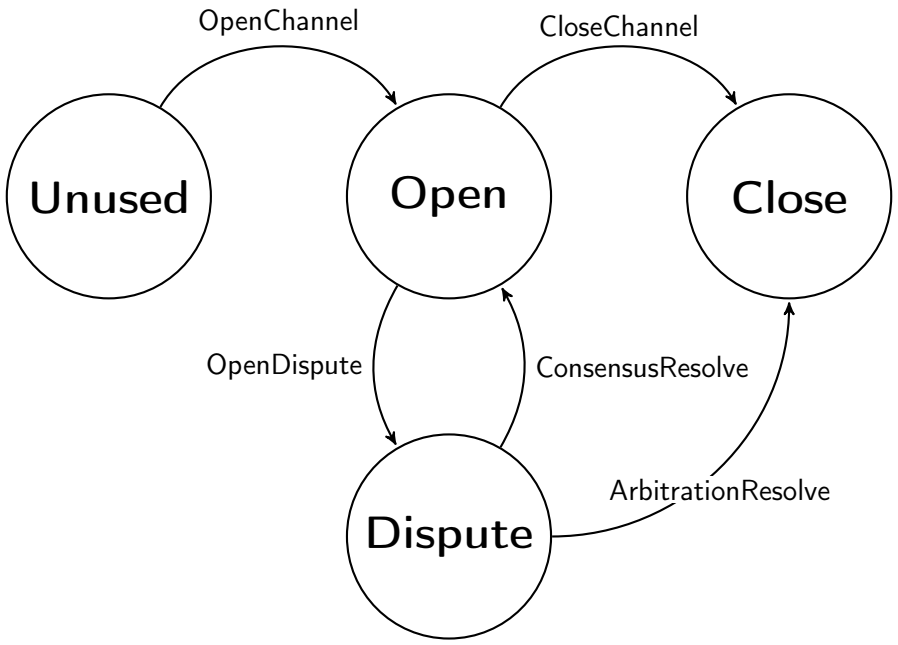
To coordinate different actions within the channel, GCCS must be able to recognize approvals from both parties. If a channel state signed by Player and by Dealer is received, GCCS considers that the participants have reached a consensus. Note that the sender does not matter in this case. To reduce

transaction costs, receiving one signature may suffice if the transaction itself is sent by the other party (i.e. approval is confirmed by sending).

Channel Status is a global variable defining the current channel status. The following status values are defined:

- *Unused* - channel wasn't opened before;
- *Open* - channel open, game in progress;
- *Close* - channel was previously used and is now closed;
- *Dispute* - channel is open, dispute in progress.

The chart below illustrates the channel life–cycle from state to state.



The dispute state is the same regardless of the underlying cause. After recording this state, the channel can either go back to open or closed depending on parties' actions. The channel goes back to open if parties are able to agree upon a new state. If the dispute is resolved through smart–contract arbitrage, the channel moves to the closed state.

Now we can say that a *Game Channel* is a channel γ such that Player and Dealer use the protocols described in this paper.

3 Game Channels

In this section we give a detailed coverage of protocols that allow two parties to open a game channel, play a game, close the channel and get rewards, without any risk of counterparty fraud. Also, we are going to consider the dispute resolution mechanism.

Let dk and pk denote, respectively, Dealer's and Player's ECDSA keys.

Table 1: The names of variables and their meanings (Solidity implementation)

Name	Type	Descriprion
channelId	bytes32	Unique channel identifier
playerAddress	address	Player's ethereum-address
dealerAddress	address	Dealer's ethereum-address
gameContractAddress	address	Ethereum-address of the game
playerBalance	uint256	Player's deposit value
dealerBalance	uint256	Dealer's deposit value
openingBlock	uint256	Information identifying when a message sent
RSAfingerprint	bytes32	Merkle-tree fingerprint of the RSA public key
gameData	bytes	Game process data
round	uint256	Round number of the game session
bet	uint256	Player's bet
seed	bytes32	Random seed for PRNG
flag	bool	Closing flag
maxNumber, minNumber	uint256	Boundaries of random numbers in the game

3.1 Opening a channel

A Player initiates the channel open event. To open a channel, its participants have to agree upon a specific initial state and confirm it with their signatures. Then the transaction with the state and participant signatures is sent to the smart contract that verifies the validity of the data. The smart contract creates a unique channel ID generated according to the following formula:

$$channelId = H(gameContractAddress, playerAddress, dealerAddress, playerBalance, dealerBalance, openingBlock, RSAfingerprint).$$

Note that either participant may send the opening transaction. For simplicity, let's assume that it is sent by the Dealer. The message exchange sequence is specified in the 1 protocol.

Protocol 1 Opening a channel

1. Player sends message containing amount of tokens for Player's deposit to Dealer.

$$initial\_message = (playerAddress, playerBalance, dealerAddress, gameContractAddress)$$

2. Dealer generates the public RSA key $RSA\_public\_key = (N, e)$ and calculates the $RSAfingerprint$. Then, Dealer generates the following messages:

$$open\_message = (playerAddress, dealerAddress, playerBalance, dealerBalance, openingBlock, gameData, RSAfingerprint, gameContractAddress)$$

$$dealer\_signed\_message = ECDSA.Sign_{dk}(open\_message)$$

3. Dealer sends the following data to Player:

$$(RSA\_public\_key, open\_message, dealer\_signed\_message)$$

4. Player receives the message, checks data in it and then signs the $open\_message$ and sends it back to Dealer.

$$player\_signed\_message = ECDSA.Sign_{pk}(open\_message)$$

5. If the $player\_signed\_message$ is valid, Dealer calls the $openChannel$ smart contract function with the following data:

$$(open\_message, dealer\_signed\_message, player\_signed\_message)$$

```
function openChannel(
    playerAddress,
    dealerAddress,
    playerBalance,
    dealerBalance,
    openingBlock,
```

```
    gameData,
    RSAfingerprint,
    gameContractAddress,
    dealer_signed_message,
    player_signed_message
)
```

6. The contract verifies validity of the received data. If valid, it is assumed that the both parties approved opening the channel. Then the contract generates $channelId$ and freezes the funds of the both parties for the game. The channel status changes to $Open$.

Example 3.1. Let's assume that Bob runs a casino. Alice wants to use Bob's service to play roulette, which is available in the list of games. First Alice allows the game contract to transfer 100 of Alice's tokens to later use them as a deposit. Then she sends the following message to Bob:

Alice: $0xde8456...$ — $(0xde8456..., 100, 0x87ff5a..., 0x8a4654...)$ → — Bob: $0x87ff5a...$

Bob analyzes the message and agrees to carry out a game. Bob allows the game contract to transfer 5000 of Bob's tokens to later use them as a deposit. Then he responds to Alice:

Alice: $0xde8456...$ — $(N, e, 0xde8456..., 0x87ff5a..., 100, 5000, openingBlock, gameData, RSAfingerprint, 0x8a4654..., Bob\_signature)$ ← — Bob: $0x87ff5a...$

Alice checks the Bob's message and, making sure it is valid, signs it on her part and replies.

Alice: $0xde8456...$ — $(0xde8456..., 0x87ff5a..., 100, 5000, openingBlock, gameData, RSAfingerprint, 0x8a4654..., Alice\_signature)$ → — Bob: $0x87ff5a...$

After checking Alice's signature for validity, Bob sends both their signatures to the contract along with the channel data.

Bob: $0x87ff5a...$	$(0xde8456..., 0x87ff5a..., 100, 5000,$ $openingBlock, gameData,$ $RSAfingerprint, 0x8a4654...,$ $Bob\_signature, Alice\_signature)$ $\longrightarrow$	Contract: $0x8a4654...$

The smart contract checks the validity of both the signatures and the balances of the participants. Then the contract locks the participant deposits. The channel is now open.

3.2 Interaction within the channel

Once the channel is open, the whole gambling process is divided into rounds. In each round a player makes specific game-related decisions (e.g. makes a bet) and sends them to the dealer with a random seed. On the basis of this seed, the dealer then computes the game result. If the player accepts the result as fair, the next round begins. The process of interactiing within a single round is defined by the 2 protocol.

Note that every time the channel state is updated, game-related-data is also recorded on the blockchain. This allows us to maintain necessary game–related statistics and to distribute rewards between the casino and the game developer according to the actual bets.

Example 3.2. Continuing with the example above, Alice places her bet of 10 tokens on red, generates the relevant message and signs it:

Alice: $0xde8456...$	$(channelId, 10, 1, gameData,$ $qw2ert5t, Alice\_signature)$ $\longrightarrow$	Bob: $0x87ff5a...$

Upon the receipt of Alice's message, Bob starts computing the round result. To do it, he calculates the hash from Alice's data and signs it via a RSA signature. The result is $0x5r43c...$ Then, on the basis of the result obtained, Bob generates a random number that determines the game result. Let's assume this number is 14. It is on red, so Alice wins the round. Bob generates a message for Alice and signs it: $update\_channel\_message = (channelId, playerBalance, dealerBalance, gameData, round)$

Alice: $0xde8456...$	$(14, 0x5f43c..., channelId, 110,$ $4990, gameData, 1, Bob\_signature)$ $\longleftarrow$	Bob: $0x87ff5a...$

Protocol 2 Messaging in the channel

1. Player generates the *seed* and the following messages:

$$seed\_message = (channelId, bet, round, gameData, seed)$$
$$signed\_seed\_message = ECDSA.Sign_{pk}(seed\_message)$$

   then sends data to Dealer.

2. Dealer checks the $signed\_seed\_message$, $seed\_message$ and computes:
   $V = H(seed\_message)$
   $S = RSA.Sign(V)$
   $S_{hash} = H(S)$
   $gameRange = maxNumber - minNumber + 1$
   while $S_{hash} \geq \left\lfloor (2^{hash.size} - 1)/gameRange \right\rfloor \cdot gameRange$ **do**
   $S_{hash} \leftarrow H(S_{hash})$
   end while
   $L = (S_{hash} \bmod gameRange) + minNumber$

   Applying the game logic to the resulting value, Dealer obtains the result for the round.

3. Dealer signs the new channel state

$$update\_channel\_message =$$
$$(channelId, playerBalance, dealerBalance, gameData, round)$$
$$dealer\_signed\_message =$$
$$ECDSA.Sign_{dk}(update\_channel\_message)$$

4. Then Dealer sends the following message to Player:

$$message = (L, S, update\_channel\_message,$$
$$dealer\_signed\_message)$$

5. Player makes sure that the number L, the game result and new participants' balances are calculated correctly.

6. Player sends the *update_channel_message* and its signature back to Dealer.

$$player\_signed\_message = ECDSA.Sign_{pk}(update\_channel\_message)$$

7. If the player or the dealer want to update the on–chain state of the game, they call the *updateChannel* function of the smart contract:

```
function updateChannel(
    channelId ,
    playerBalance ,
    dealerBalance ,
    gameData ,
    round ,
    dealer_signed_message ,
    player_signed_message
)
```

Alice approves the received data upon checking it. She signs the message and sends it to Bob.

Alice: $0xde8456...$ $\xrightarrow{(channelId, 110, 4990, gameData, 1, Alice\_signature)}$ Bob: $0x87ff5a...$

At the request of either party, the message and both signatures can be sent to the contract to update the data stored in it.

Bob and Alice exchange messages until either of them decides to close the channel for some reason.

3.3 Closing the channel

If either party wants to close the channel they initiate the relevant query. There are following reasons for this:

1. Player or Dealer voluntary decides to stop gambling. The 3 protocol defines the sequence of actions for this scenario.

2. Either participant has zero balance and/or lacks tokens for the next bet. If either party sends the closure query, and the other accepts it, the 3 protocol is applied. Otherwise the 4 protocol is used.

3. The Channel has expired. Note that participants have to monitor the channel validity period on their own. Once it is expired, all smart contract functions become unavailable, except for the $closeBYTime$ function. This is the 5 protocol.

4. If either party stops responding to messages, the 6 protocol is applied.

5. Data forgery by either participant. If an attempt to upload forged data to the channel is detected, the contract returns an error message. Thus, invalid data cannot get to the blockchain, and can only be stored locally by participants. When either player suspects the other of fraud, the 6 protocol is applied. Note that for the smart contract this case is similar to the previous one as far as the contract logic is concerned.

Protocol 3 Consented channel close

---

1. A party willing to close a channel sends their current round state to the other participant.

$$message = \\ (channelId, playerBalance, dealerBalance, gameData, round)$$

2. If the other participant accepts this state, this participant returns the following message with a signature:

$$close\_message = \\ (channelId, playerBalance, dealerBalance, gameData, round, flag) \\ party2\_signed\_message = ECDSA.Sign(close\_message)$$

3. The first party then validates the received message, and, if approved, signs the $close\_message$ as well.

$$party1\_signed\_message = ECDSA.Sign(close\_message)$$

---

4. Dealer holding the *party*1_*signed_message* and *party*2_*signed_message*, sends them to the contract with the *close_message*.

```
function closeByConsent(
    channelId,
    playerBalance,
    dealerBalance,
    gameData,
    round,
    party1_signed_message,
    party2_signed_message
)
```

5. The contract validates received signatures. Then makes sure that amounts frozen in the contract equal tokens that the player and the dealer intend to withdraw.

6. If all the conditions are met, the contract initiates the *closeChannel*(*channelId*) function and distributes tokens between parties according to the data received.

7. The contract then deletes the channel via the *removeChannel*(*channelId*) function. The contract changes the channels status to *Close*.

Protocol 4 Low balance channel closure

1. The first party makes a request to the smart contract *updateChannel* function and sends it the latest signed state indicating that either player has a zero token balance or insufficient balance to bet.

2. The *updateChannel* function updates the stored data and simultaneously checks player balances. If zero or insufficient balance is confirmed, the channel is closed and removed (see items 6 and 7 for the 3 protocol).

Protocol 5 Expiration closer

1. Either party calls the $closeByTime(channelId)$ function. The function checks whether or not a dispute was initiated during its execution.

2. If yes, the $closeByDispute(channelId)$ function is executed; it interprets the channel state in favor of Player, giving them the highest possible reward provided in the game logic.

3. The channel is closed and removed (see items 6 and 7 for the 3 protocol).

Protocol 6 Nonresponse/data forgery closery

1. A participant uploads the latest approved state to the smart contract via the *updateChannel* function.

2. The participant then attempts to open a dispute via the following contract function:

```
function openDispute(
    channelId,
    round,
    bet,
    gameData,
    seed,
    player_signed_message
)
```

   where $player\_signed\_message = ECDSA.Sign_{pk}(channelId, round, playerBalance, gameData, seed)$.

3. The smart contract verifies the data received from the participant; if it is valid, a dispute is opened. The channel status changes to *Dispute*.

4. Participants then have $t1$ time blocks to call *updateChannel* to provide a newer valid state; there is also another function is *doubleSign* (only available to the Dealer). The first function is followed by step 4.1 of the protocol, the second is followed by step 4.2.

4.1. After checking the validity of the data, *updateChannel* compares the round number when the dispute was opened with the round number in the updated state. If the difference is more than one, the dispute opener loses their deposit. The channel is closed and removed (see items 6 an 7 for the 3 protocol). Otherwise the dispute is removed after the channel update, and the channel state reverts back to *Open*. Note that it does not matter which party provides the channel update.

4.2. When *doubleSign* is called, the smart contract checks whether or not a player sent two different signed messages with the same round number. If so, Player loses all their deposit. The channel is closed and removed (see items 6 and 7 for the 3 protocol).

```
function doubleSign(
        channelId,
        round,
        bet,
        gameData,
        seed,
        player_signed_message
    )
```

5. After $t1$ time blocks the $t2$ time window opens, allowing parties to update the channel state or call the *doubleSign* function; otherwise Dealer can call the *resolveDispute* function from the contract. If none of the options is selected for the allocated $t1 + t2$ time span, the 5 protocol applies. In this case there are no additional steps.

```
function resolveDispute(
        channelId ,
        N,
        e
        S
    )
```

5.1. The *resolveDispute* function checks the dealer RSA public key and verifies the RSA signature S. If incorrect, the function call is aborted.

5.2. The $resolveDispute$ function calls the $runGame(channelId, playerBalance, S)$ function.

5.3. The $runGame$ function verifies the game logic and withdraws player balances.

5.4. The channel is closed and removed (see items 6 and 7 for the 3 protocol)

Example 3.3. Continuing with the example, let's assume that Alice bets 20 tokens in round 5 and sends the data to Bob in a message.

Alice: $0xde8456...$ $\xrightarrow{(channelId, 20, 5, gameData, qwrtty)}$ Bob: $0x87ff5a...$

But, Bob has just had a connection failure and cannot respond to Alice. Alice gets no response and uploads the last agreed upon state to the channel; then she opens a dispute with a new request. The contract allocates $t1 + t2$ time blocks to Bob so that he can upload a newer state to the channel. Suppose that after $t1 - 1$ blocks Bob restores the connection in time to provide a new state with game results, via the $updateChannel$ function. The dispute is then removed, and Alice and Bob go on gambling.

When the channel lifespan at round n comes close to expiration, Bob decides to close the channel. He requests Alice's approval to close the channel with the current state.

Alice: $0xde8456...$ $\xleftarrow{(channelId, 85, 5015, gameData, n)}$ Bob: $0x87ff5a...$

Alice checks the received message and sends a message with the data required to close the game, along with a signature, to Bob.

Alice: $0xde8456...$ $\xrightarrow{(channelId, 85, 5015, gameData, n, true, Alice\_signature)}$ Bob: $0x87ff5a...$

Bob, in turn, checks the received data and, upon making sure the data is valid, signs the data and sends it with both signatures to the contract.

$$\text{Bob: } 0x87ff5a... \xrightarrow{(channelId, 85, 5015, gameData, n, true, Bob\_signature, Alice\_signature)} \text{Contract: } 0x8a4654...$$

Suppose that the signature validity condition is then met for both signatures. Therefore, upon checking the received data, the contract closes the channel and removes it.

Example 3.4. Suppose that the same time there is another player gambling with Bob, Mallory, who is a fraudster. There is an open channel between them. Mallory makes a bet of 2 tokens on red, generates a message and sends it to Bob:

$$\text{Mallory: } 0xca62a232... \xrightarrow{(channelId, 2, 1, gameData, vg345)} \text{Bob: } 0x87ff5a...$$

Bob computes the result for Mallory. It is 20 black; Mallory loses his bet and Bob sends him the relevant message:

$$\text{Mallory: } 0xca62a232... \xleftarrow{(20, 0x53c..., channelId, 3, 502, gameData, 1, Bob\_signature)} \text{Bob: } 0x87ff5a...$$

Unhappy with the result, Mallory tries to send back to Bob a state with forged data in which the result favors him.

$$\text{Mallory: } 0xca62a232... \xrightarrow{(channelId, 7, 498, gameData, 1, Malory\_signature)} \text{Bob: } 0x87ff5a...$$

Upon checking Mallory's message, Bob detects invalid data, loads the last valid state and initiates a dispute via the *openDispute* function. The contract allocates $t1 + t2$ time bocks to Mallory to provide a newer state signed by the both parties. Mallory retries the forged state, uploading it to the channel via the *updateChannel* function. The contract checks the data, detects that it is invalid and returns an error. As a result, Mallory fails to provide a newer state to the channel within the allocated $t1$ time span. Now Bob launches the *resolveDispute* function. This function, in turn, makes sure that Bob's data is valid and calls the *runGame* function that defines the game logic. The *runGame* function checks the gambling process and concludes that 2 tokens have to be removed from Mallory's account and deposited to Bob's account.

The channel is then closed and removed.

4 Modifications

In this section we introduce some modifications to the basic protocol. First, we will cover conversion of the game channel into a version without the Dealer role, with two equal players. The other modification allows connecting a third party to the channel that will not directly participate in the game, but will receive published channel states with participant signatures.

4.1 Two Players Case

Some games (e.g., some dice variations) involve two equal players; a casino takes no part in the process. The original protocol version covered in the Game Channels section can be used for these games. But for completeness, we want to suggest an alternate protocol version based on the *Threshold Signature Scheme*. Obviously, the threshold signature must have the uniqueness property (e.g. TBLS or TRSA [22] signatures).

Let's redeclare the Player role as $Player1$, and the Dealer role as $Player2$. For the two player scenario an altered version of the 1 protocol is applied to open the channel. The RSA signature is replaced with the selected threshold signature scheme: τ is replaced by the $\tau.PartSign()$ algorithm. This algorithm takes as input a message m and outputs a partial signature of the message for any participant. Item 2 is replaced with the DKG protocol, which allows each party to have a part of the private key and a shared public key. Just like in the original protocol, the channel state changes to *Open* when the channel receives two valid participant signatures.

Once the channel is open, a round involving interaction of two peer parties takes place under the new protocol at the 24 page. Obviously, upon completion of a round each party gets a signed game result and a signed message from the other party with the same data in it. Dispute can be initiated in the case of data discrepancy.

To close a channel, the same methods are used as specified in subsection 3.3. The protocols covered in that section are compatible with this modification with a minor alteration. Now bet is considered to be committed once both participants have posted their seeds. Also, the $resolveDispute$ and $doubleSign$ functions are available to either participant. (see the 5 item for protocol 6).

Protocol 2.1 Messaging in the channel

1. Participants generate and exchange messages of the following type:

$$seed\_message = (channelId, bet, round, gameData, seed)$$
$$signed\_seed\_message = ECDSA.Sign(seed\_message)$$

2. Participants verify the received $signed\_seed\_message$ and carry out the following calculations:

$aggregate\_seed\_message =$
$= seed\_message$ (from Player1) || $seed\_message$ (from Player2)
$V = H(aggregate\_seed\_message)$
$S = \tau.PartSign(V)$

3. They then exchange messages with their respective signature fragments.

$$message = (S, round, gameData, player1Balance, player2Balance)$$

4. Players verify whether the S number is calculated correctly. If this condition is met, the next step follows immediately.

5. To compute the game results, the players holding two segments of the signature merge them into a single $aggregate\_S$ that depends on the selected τ and on the DKG protocol.

$S_{hash} = H(aggregate\_S)$
$gameRange = maxGame - minGame + 1$
while $S_{hash} \geq \lfloor 2^{hash.size}/gameRange \rfloor \cdot gameRange$ **do**
    $S_{hash} \leftarrow H(aggregate\_S_{hash})$
end while
$L = (S_{hash} \bmod gameRange) + minGame$

6. Players exchange the game results and verify the data.

$$message = (channelId, player1Balance, player2Balance, gameData, round)$$
$$signed\_message = ECDSA.Sign(message)$$

7. If needed, either player may use the two signatures of the result message to update the channel state.

Note that our two player modification is described for informational purposes only. We recommend using the original version (with the additional step for the Dealer's bet), as it has same security properties, but is more efficient in terms of speed.

4.2 Third Party Observer

Pisa [13], as mentioned above, allows connecting a "watcher" to a channel. If one of the actual players happens to get disconnected, the watcher will be able to stand in for that player in interacting with the smart contract. The design of the Game channels allows us to easily apply a similar approach and connect a third participant. To that end, any time participants approve some state, they post it with their respective signatures. The third participant listens to the channel and can then update the smart contract state using these messages. Note that the third participant needs no lock or verification within the smart contract.

This approach can be useful in the design of a platform where players and dealers meet. In this case a platform can assume responsibility for all contract requests, reducing player costs incurred from additional transactions. The downside of this approach is increased system centralization. Requests to a smart contract via the platform can be implemented as optional, not mandatory, functionality.

Acknowledgements

We would like to thank research scientist Mark Reynolds for proofreading and feedback.

We also thank Maria Kondorskaya for her help with translation.

And we thank Vlad Gluhovsky for the basic specification of the Signidice algorithm and help in creation of this solution.

References

[1] Bitcoin Wiki: https://en.bitcoin.it/wiki/Payment_channels

[2] Dan Boneh, Ben Lynn, and Hovav Shacham. *Short signatures from the Weil pairing*. Computer Science Department, Stanford University https://www.iacr.org/archive/asiacrypt2001/22480516.pdf

[3] Tom Close and Andrew Stewart. *Force-Move Games*. June 21, 2018.

[4] Jeff Coleman. *State Channels.* https://www.jeffcoleman.ca/state-channels/. Nov 6, 2015

[5] Jeff Coleman, Liam Horne, and Li Xuanji. *Counterfactual: Generalized State Channels*. https://l4.ventures/papers/statechannels.pdf. June 12, 2018.

[6] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Daniel Malinowski. *PERUN: Virtual Payment Hubs over Cryptographic Currencies*.

[7] Don Johnson, Alfred Menezes, Scott Vanstone. *The Elliptic Curve Digital Signature Algorithm ECDSA*. February 24, 2000.

[8] Amit Kumar Jaiswal. *Parsec: A State Channel for the Internet of Value.* https://arxiv.org/pdf/1807.11378.pdf

[9] Daniel Kraft. *Game Channels for Trustless Off-Chain Interactions in Decentralized Virtual Worlds*. 2016.

[10] Jeremy Longley and Oliver Hopton. *Funfair technology roadmap and discussion*. https://funfair.io/wp-content/uploads/FunFair-Technical-White-Paper.pdf. 2017

[11] Alex Lunyov, Johann Barbie, Konstantin Korenkov, Mayank Kumar. *Multiparty State Channels Enabling Real-Time Poker on Ethereum*. https://www.acebusters.com/files/acebusters_yellowpaper.pdf

[12] Ian Macalinao. *Why EOS will overtake Ethereum in high performance smart contracts*. https://ian.pw/posts/2017-12-08-why-eos-will-overtake-ethereum-in-high-performance-smart-contracts.html

[13] Patrick McCorry, Surya Bakshi, Iddo Bentov, Sarah Meiklejohn, Andrew Miller. *Pisa: Arbitration Outsourcing for State Channels.* IC3, University College London, University of Illinois at Urbana Champaign, Cornell University. https://www.cs.cornell.edu/~iddo/pisa.pdf

[14] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, Patrick McCorry. *Sprites: Payment Channels that Go Faster than Lightning.* UIUC, Cornell University, Microsoft Research, Newcastle University. https://allquantor.at/blockchainbib/pdf/miller2017sprites.pdf

[15] Joseph Poon and Thaddeus Dryja. *The bitcoin lightning network: Scalable off-chain instant payments.* https://lightning.network/lightning-network-paper.pdf. 2016.

[16] Raiden specification. http://raiden-network.readthedocs.io/en/stable/spec.html. 2018.

[17] R.L. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* 1978.

[18] Mark Schwarz. *Transaction Speeds: Which Crypto Is the Fastest?* https://www.abitgreedy.com/transactionspeed/transaction-speed. June 26, 2018.

[19] *The Signidice Algorithm.* https://github.com/gluk256/misc/blob/master/rng4ethereum/signidice.md

[20] Josh Stark. *Making Sense of Ethereum's Layer 2 Scaling Solutions: State Channels, Plasma, and Truebit.* https://medium.com/l4-media/making-sense-of-ethereumslayer-2-scaling-solutions-state-channels-plasma-and-truebit-22cb40dcc2f4

[21] *State Channels.* https://github.com/aeternity/protocol/blob/master/channels/README.md

[22] C. Stathakopoulou, C. Cachin. *Threshold Signatures for Blockchain Systems.* April 4, 2017.

[23] Team Keccak. https://keccak.team/keccak.html.

[24] Xaya Game Library and Mover. https://github.com/xaya/libxayagame.