cf-specs Documentation

Counterfactual Contributors

Apr 22, 2019

Contents:

CHAPTER 1

Introduction

The Counterfactual Protocol is a state channels based protocol for off-chain blockchain-based applications. The protocol has been designed for general purpose state channels: an important technique for reducing fees for blockchain users. State channels allow users to interact with each other without paying blockchain transaction fees and with instant finality. Channelization is the only technique that securely realises the latter property.

In the protocol, participants exchange cryptographically signed messages through an arbitrary communication channel. These messages are pre-signed transactions that distribute the blockchain state or perform other tasks necessary to the channel's correct resolution. Next, participants deposit blockchain state into an n-of-n multisignature wallet referenced by the transactions. New cryptographically signed state updates having a special relationship to the original commitments can now be used to change the state and / or assets controlled by the multisignature wallet.

Through a challenge-response mechanism, on-chain contracts implement methods for participants to ensure that the latest signed valid state update that pertains to their commitment can be submitted to the blockchain, guaranteeing correct resolution of the state for all users adhering to the protocol.

Counterfactual uses a generic system of Ethereum smart contracts to support arbitrary conditional transactions of blockchain state owned by a multisignature wallet. For a full explanation of the contracts layer, please read the contracts subsection.

1.1 Framework Design Goals

The Counterfactual framework is still a work in progress. Its current design (and future roadmap) are driven primarily by the following criteria:

1.1.1 Minimized on-chain footprint

We don't want to put anything on the chain that doesn't need to be. We aim to make a generic multisignature wallet the only necessary on-chain object for a state channel.

1.1.2 Maximized privacy

We want to achieve a level of privacy where state channel operations are indistinguishable from other common types of on-chain activities. Using a state channel should not reveal any information about the applications that are being used, the state being used within them, or even the fact that a state channel is being used at all. As a first step towards preserving this property, we assume that the on-chain component is a generic multisignature wallet which looks the same as any other multisignature wallet on Ethereum. In the future we expect that stricter levels of privacy will be enabled by various zero knowledge constructions, and that those will fit best when applied in similarly general, abstract ways that fit neatly with this approach.

1.1.3 Ease-of-use

We want channels that can be easily incorporated into new applications without the requirement that their developers also be state channel experts. To provide at least one such simple method for developers to utilize within our framework, we have created an abstraction for state-machine-based channel applications, or "Apps". This class of "App" consists of simple stateless contracts which define a state machine, including valid transitions and turn-taking logic. Although state-machine-based "Apps" are an intentionally restricted subset of state channel functionality, they nonetheless enable developers to deploy a wide range of channelized applications without butting up against the often complex and subtle *limitations* of state channel design. As the protocol develops further more complex functionality will continue be added, allowing easy utilization of increasingly advanced techniques by developers making use of the Counterfactual framework.

1.1.4 Parallel operations

We want to support multiple parallel operations inside of a single channel that do not interfere with each other. We have designed these simple state-machine-based "Apps" to maintain control of the state assigned to them in a fashion completely independent of each other. Typical operations like installing new applications, uninstalling old applications, and updating applications are all fully parallelized operations within the protocol.

1.1.5 Upgradeable

We want to support deploying or upgrading channel designs without requiring the user to make a single on-chain operation. There are multiple techniques which are specifically anticipated in the current design. For the purposes of *trustless* off-chain upgradability, we are able to support counterfactually instantiated smart contracts as applications. To upgrade a contract trustlessly, state channel participants can simply agree off-chain to a new version of bytecode for their application. At the cost of certain additional trust assumptions, state channel participants could also use an application that is defined using ZeppelinOS's upgradeable contracts or a similar method.

1.1.6 Standardized

We want to establish clear standards for how all of these generalized state channels will fit together into a global, multi-blockchain network where any user can easily connect to any other. To achieve this goal, we work closely with great researchers from Celer, Magmo, Ethereum Research and several others. We hope to amplify these efforts to work towards blockchain standards for off-chain channelized applications more broadly.

1.2 Protocol Design Goals

1.2.1 One round trip communication

The protocol has been optimized to require the minimum number of round trips possible for secure updates to the off-chain state. This has effectively led to the use of multiple transactions being bundled inside a single transaction using the `MultiSend` contract built by Gnosis.

The goal of having one round trip for each protocol execution is purely an optimization and it is the case that "uncompressed" variations do exist that are equally as secure but more costly in communication overhead. This is currently up for discussion.

1.2.2 Constant sized communication

The number of messages an message sizes for an operation are independent of

- Total number of active off-chain applications
- Total number of inactive off-chain applications
- Total number of sequential state updates to an application

That is to say, the design aims for **parallelizability** in general, ensuring that historical use of the protocol does not impact the size of messages being transmitted on future use of the protocol.

1.2.3 O(1) response to stale state

It is possible to arrive at a state where any placement of stale state on chain can be responded to with a single transaction of constant size, in particular, independent of number of active or historical apps. This goal is to ensure that any kind of inevitable griefing vectors that are impossible to fully disqualify off-chain are resolvable with the minimum amount of cost to the person being griefed on-chain.

1.3 Protocol Structure

A protocol consists of the following components:

- **Exchange**. The protocol exchange is the series of messages exchanged between all parties in the state channel, as well as dependencies between messages.
- **Message**. A message is the set of information that must be exchanged by the parties to recreate and validate the commitment signatures and associated transactions that those signatures enable. Each protocol may in general contain multiple message types.
- **Commitments**. A protocol produces one or more commitments. Both the signatures and the data contained within the commitments must be stored.

  Note: Messages are represented as JSON-encoded objects; the transport layer should be able to reliably send and receive such messages.

  Note: Some data required for signature verification is *not* present in these messages but is deterministically generated by each party. For example, `setNonce` requires a salt, which is a monotonically increasing counter on the number of applications in a channel.

1.3.1 Commitment Structure

- **Signature**. The resultant data generated by computing a cryptographic signature over some hash representing the transaction to be executed.
- **Data**. Supplementary data that that allows any party in the protocol to reconstruct the hash and thus verify the signature.
- **Transaction Digest**. The transaction digest is the hash that is signed by each party, enabling the protocol's transaction to be executed on-chain. The calldata, if present, is used to generate the digest.
- **Transaction**. The transaction is the `(to, val, data, op)` tuple that a given protocol allows one to broadcast on-chain. These transactions enforce commitments created from the calldata and signature digests, manifesting the off-chain counterfactual state into the on-chain reality.

CHAPTER 2

State Channel Applications

When we discuss building off-chain applications in general, we usually reference easy-to-understand examples such as payment channels, turn-based games like Tic-Tac-Toe, or well understood blockchain use cases like mixers. For each of these examples, we attempt to isolate their core functionality inside of a single logical container inside the Counterfactual framework. We label these containers as **Apps** inside the framework. APps have the following properties at a minimum:

- A fixed set of participants / users
- An encoding type for its state
- A resolution function based on the state

The most basic type of App is a 2-person ETH payment channel. In that case we have:

- Alice and Bob as the users
- An encoding type of `tuple(address aliceAddr, address bobAddr, uint256 aBal, uint256 bBal)`†
- The resolution function sends `aBal` Wei to `aliceAddr` and sends `bBal` Wei to `bobAddr`

A slightly more complicated example would be a Tic-Tac-Toe game:

- Alice and Bob are the users, Alice is X and Bob is O
- An encoding type of `tuple(uint8[9] board, address playerX, address playerO)`
- Sends `playerX` the maximimum amount if X won, or `playerO` if O won, or splits the amount in half if a draw

There is, of course, an important difference in the example of a Tic-Tac-Toe game though. That difference is that in the Tic-Tac-Toe game, the final resolution of the state of the application is not always defined based on the current state of the application. In the payment channel example, if one user were to become unresponsive it is is easy to see how the resolution would play out; Alice would receive `aBal` and Bob would receive `bBal`. In Tic-Tac-Toe, however, if the game is not finished yet, there are some moves left to be made to reach a "terminal" state which can then be resolved: X having won, O having won, or a draw to be specific.

To express this important difference we introduce some additional functionality that can be implemented for a state channels based application:

- A function which defines how state can "progress" given an action
- A function to define whether an action can be legally taken by a participant
- A function to determine is a state is indeed one that could be considered "terminal"

For Tic-Tac-Toe then these can be expressed as:

- Allowed actions are to place an X on the board or place an O on the board
- `playerX` may place an X if it is X's turn based on the `board` state (and vice versa)
- The state is "terminal" if there are 3 in a row of X's or O's on the board or the board is full

In the next sections, we define how in Counterfactual we define, resolve, and progress state.

2.1 Defining State

A state channel is fundamentally about progressing a single state object. Therefore, in the Counterfactual framework, we center everything around a single `bytes` object. For the sake of performing computation on this object (e.g., evaluating if there is a row of X's on a Tic-Tac-Toe board) we interpret it as an ABI-encoded value of a specific type using the built-in `ABIEncoderV2` language feature. THIS allows developers to define their state structures using a `struct` definition and encode and decode these objects as needed. For example, in our payment channel from above we might have the following object:

```
struct ETHPaymentChannelState {
  address alice;
  address bob;
  uint256 aBal;
  uint256 bBal;
}
```

2.2 Progressing State

As has been mentioned before, some kinds of applications require a way of progressing some state to a "terminal" state through a series of allowed actions. In these cases, we adopt the model of a state machine that consists of logical states and allowed actions (transitions between states); a "terminal" state is simply one from which there does not exist any outgoing edge (i.e., an "allowed action").

In Counterfactual, if an action wishes to allow its state to be unilaterally progressable, we require the definition of a function that **applies an action to a state to produce a new state** *in addition to* **a function that determines if an action can be taken by a particular turn taker**. As you will see in the *adjudication layer* section of these specifications, these functions are important in handling on-chain challenge scenarios.

The ultimate purpose of these functions is to ensure the following:

- It should always be extremely explicit what the exact rules of the state channel application that all parties are abiding by are
- There should always be a single logical turn taker for any given state (*concurrent state updates* are disallowed)
- It should be possible, in the least number of on-chain transactions, use the adjudication layer to fairly resolve a state channel application (without unnecessary gas expenditure)

Here is a helpful diagram for visualizing the nature of such an `applyAction` function:

2.3 Resolving State

A state channel application defines a resolution. This is a critical concept usually because the resolution can be tied to interesting economic parameters that create the incentives for the behaviour of users in the application to begin with. For example, users will remain online and play a game of Tic-Tac-Toe because they know the rules of the game are deciding who will take home some financial reward. In the Counterfactual framework, this reward is defined in terms of an internal transaction that is executed by the multisignature wallet that is the holder of the state deposits.

When writing a state channel application presently, we require that the resolution of an application be directly tied one-to-one to some particular state that is being progressed. For example, when defining a game of Tic-Tac-Toe, we ask that the resolution function which checks for the winner specifically return a data structure that can be interpretted for sending ETH to the user that won.

Here is a diagram that shows how a resolution looks presently for a game of Tic-Tac-Toe:

Note: This is subject to an improvement that will be added soon that will de-couple an application from the exact transaction that occurs upon resolution to an interpreter pattern that will allow for multiple assets to be re-used on the same state machine logic. For example, a Tic-Tac-Toe game either returns a full-amount for one person or divides an amount in half; those are the only two options. These two options can be mapped to ETH, ERC20 tokens, or other assets which are fungible in single or half quantities.

2.4 AppDefinition

To address all of the above requirements of state channel applications, we introduce an interface called an `AppDefinition` which **implements the logic of an application in the EVM**. The `AppDefinition` interface is implemented by a developer interested in writing a state channels based application that the Counterfactual project supports in the rest of the framework (e.g., in the *adjudication layer*).

2.4.1 Resolution Function

There is a single function which *must* be implemented in the interface. This function provides the resolution functionality discussed above. The function signature is:

```
function resolve(bytes memory, Transfer.Terms memory) public view returns (Transfer.
↪Transaction);
```

The first argument of type `bytes memory` is an internally-referencable state object for the `AppDefinition`. For example, in the case of a payment channel it must be considered as the ABI-encoded version of a structure encoding the type from above. This means that you will likely want to use `abi.decode` inside of the `resolve` method to decode the bytes into a usable struct object. In our payment channel example this would look like:

```
function resolve(bytes memory encodedState, Transfer.Terms memory terms)
  public
  view
  returns (Transfer.Transaction)
{
  AppState state = abi.decode(encodedState, (ETHPaymentChannelState));
  // state.aBal, state.bBal, state.alice, state.bob
}
```

> **NOTE**: The `Transfer.Terms` argument and required return type `Transfer.Transaction`, although necessary at the moment, are being changed in a pull request to be merged very shortly from the time of this writing (#1263) so the descriptions are omitted from this document for now.

In addition to the `resolve` method, the `AppDefinition` interface also allows for the optional definition of the following methods which can either all be implemented or none implemented.

2.4.2 Turn Taking Function

To identify who is uniquely allowed to progress from one state to the next, a turn taking function can be implemented which returns the specific address that is expected to have signed a particular state update. It accepts two arguments: the state of the application and the array of all signing keys that have been allocated to the application. Therefore, the function must return the key in the array of signing keys that it expects to sign an update.

```
function getTurnTaker(bytes memory, address[] memory) public pure returns (address);
```

In our Tic-Tac-Toe example, this function would return the 0th-indexed key for player X and the key at index 1 for player O.

2.4.3 Terminal State Flag Function

As an efficiency gain in cases where the adjudication layer is needed, a function can be defined which declares whether or not a state can no longer be progressed. This function takes a single argument: the state of the application. It is expected to return `true` if the state is terminal or `false` if it is not.

```
function isStateTerminal(bytes memory) public pure returns (bool);
```

In Tic-Tac-Toe, the state is terminal if the game has been won or the board is full.

2.4.4 Action Application Function

Finally, the most critical function for progressing state is the `applyAction` function which as described above takes some encoded state and an encoded action and returns a new encoded state object. The *encoding and decoding* functionality provided in Solidity are helpful here.

```
function applyAction(bytes memory, bytes memory) public pure returns (bytes memory);
```

In Tic-Tac-Toe, this function would place the X or the O on the board based on the action type if the position on the board is not already filled.

2.5 Footnotes

2.5.1 ABIEncoderV2

We use `ABIEncoderV2` in the framework to represent arbitrary EVM-compatible state

There are helpful functions that are offered in Solidity which can be used for encoding and decoding.

Encoding:

```
ExampleStruct state = ExampleStruct(...);
bytes encodedState = abi.encode(state);
```

Decoding:

```
bytes encodedState; // 0x.....
ExampleStruct state = abi.decode(encodedState, (ExampleStruct));
```

2.5.2 Concurrent State Updates

Presently in Conterfactual progressable state machines must be uniquely progressable by a single turn taker and must move one turn at a time. The reason for this requirement is that if two or more users were able to progress the state of an application independently, it is possible that they may enter into a conflicting state.

As an example of this, consider an application where two users can move the the (x, y) co-ordinates of two pieces on a 10x10 grid; each can move one but not the other. In this example, it is possible that if they were able to move their pieces concurrently they could enter into a state where two pieces are in the same (x, y) co-ordinate which may be disallowed in the logic of the application. In that case a race condition occurs where whoever is able to submit their transaction to the blockchain first would be able to consider their state as final and the other would be disallowed. Of course, this is a kind of behaviour we want to avoid when designing state channel applications.

There are scenarios, however, where you *want* concurrent updates and it is indeed possible to allow them. The general class of data structures that represent these types of data structures are called conflict-free replicated data types (CRDTs) and are commonly used in distributed systems.

In a future version of the `AppDefinition` interface, we hope to support a version of these kinds of objects such that concurrent state updates might be possible.

CHAPTER 3

Adjudication Layer

Counterfactual's adjudication layer uses a singleton contract called the `AppRegistry`.

The implementation that is inside of this repository has been designed to only be compatible with applications that implement the `CounterfactualApp` interface. Although this is the case, the core concepts are agnostic to the underlying interface that a state channels application might implement. A future version of the `AppRegistry` might support other types of state channel architectures such as Force Move Games, for example.

The two core concepts of the adjudication layer are **challenges** and **resolutions**. Challenges are the adjudication layer's mechanism to *learn the final state* of an off-chain application and resolutions *distribute state deposits* based on the resolution. As a concrete example, a challenge might be about the state of the board in a game of Tic-Tac-Toe and a resolution might be about who is rewarded with the 2 ETH allocated to the off-chain application.

> In some other frameworks, these two concepts can be implicitly grouped together in such a way that the resolution might be *a part of the state itself*. From a conceptual point of view, we think it is important to separate these two concepts. However, from an engineering point of view it can be more efficient to group the two together in a single state object. To be specific, if the resolution of an off-chain application is a agnostic to blockchain state (i.e., `pure`) operation on the state of the application, then it is safe to group the two together. However, if the resolution has a dependancy on an external contract or the block number (i.e., a `view` function) then the resolution *must* be separately resolved.

3.1 Challenges

A challenge that is put on-chain must result from a failure of some state channel users to follow the protocol. In most cases, it represents a failure of responsiveness.

3.1.1 Data Structure

In the `AppRegistry`, a challenge is represented by the following data structure:

```
struct AppChallenge {
    AppStatus status;
    address latestSubmitter;
    bytes32 appStateHash;
    uint256 disputeCounter;
    uint256 disputeNonce;
    uint256 finalizesAt;
    uint256 nonce;
}
```

Where `AppStatus` is one of `ON`, `OFF`, or `DISPUTE`.

Here is a description of why each field exists in this data structure:

- **`status`**: A challenge exists in one of four logical states.
  - `ON`: Has never been opened (the "null" challenge and default for all off-chain apps)
  - `DISPUTE`: Open and its timeout parameter is in the future (can be responded to)
  - `DISPUTE`: Was opened and the timeout expired (the challenge was finalized)
  - `OFF`: Was finalized explicitly

statechannel statuses

Thus, this parameter simply records which state the challenge is in.

- **`finalizesAt`**: This is the block number at which a challenge is considered to be finalized if there is no reponse from anyone that is able to respond to the challenge.
- **`latestSubmitter`**: This is a record of *who* submitted the last challenge. This is useful to ensure that if a *provably malicious* challenge was submitted, the malicious party can be punished.
- **`appStateHash`**: This is the hash of the latest state of the application. We only need to store the hash of the latest state in the event of a challenge because it is possible to accept the full state as calldata, hash it in the EVM, and then compare this to the hash and keep the full state available in calldata to be used for computation.
- **`nonce`**: This is the nonce (i.e., the monotonically increasing version number of the state) at which the `appStateHash` is versioned for a particular challenge.
- **`disputeNonce`**: It is allowed for a challenge to be responded to by issuing a new challenge. This idea is isomorphic to the familar state channels concept of "continuing the game on-chain". In this event, you want to keep track of the number of times a challenge has been "re-issued" as a new sense of versioning. We cannot simply increment the `nonce` field for reasons that are described in the section below: *on-chain progressions of off-chain state*.
- **`disputeCounter`**: A challenge *can* be unanimously cancelled by all parties. In this scenario the `disputeNonce` goes back to 0. However, the `disputeCounter` is a permanent marker of how many times a challenge has been issued and re-issued on-chain. Parties can pre-agree that if this counter ever reaches an excessively high number, to simply resolve the application at some pre-determined resolution. This is simply a special mechanism to resolve applications in cases of excessive griefing by one counterparty. For more information, I recommend reading the economic risks section of the Counterfactual paper.

Since the contract that Counterfactual relies on for managing challenges is a singleton and is responsible for challnges that can occur in multiple different state channels simultaneously, it implements a mapping from what is called an `AppIdentity` to the challenge data structure described above.

The `AppIdentity` looks as follows:

```
struct AppIdentity {
    address owner;
    address[] signingKeys;
    address appDefinitionAddress;
    bytes32 termsHash;
    uint256 defaultTimeout;
}
```

Here is a description of why each field exists in this data structure:

- **owner**: As has already been mentioned, the on-chain state deposit holder is a multisignature wallet with an `execTransaction` function on it. This field records the address of that multisig. It is used to treat any function call where `msg.sender == owner` as having achived unanimous consent.
- **signingKeys**: In addition to using `owner` to authorize a function call (whereby the signature verification is done inside the multisignature wallet contract), it is also possible to pass in signatures directly into the `AppRegistry` itself. In these cases, this field is used to validate signatures against to consider a function call as "authorized".
- **appDefinitionAddress**: This is the address ofthe app definition contract.
- **termsHash**: An application must adhere to some terms which describe what the *resolution* of the application (should it ever be challenged on-chain) would need to adhere to. The danger that requires this strict adherence is that *any developer* can write a `resolve` function for an application and this resolution will be executed in the scope of a `DELEGATECALL` on the multisignature wallet. Therefore, this property acts as a commitment by all parties to adhere to some `Terms` (which `termHash` is the hash of) to be verified against.
- **defaultTimeout**: Should the application that this data structure is describing ever be put on-chain, this property describes how long the timeout period would be if that challenge ever gets responded to on-chain. In the case of a challenge *initially* being put on chain, the timeout period is a required parameter regardless, so this field is not used in that case.

3.1.2 Initiating a Challenge

Counterparty is unresponsive. In the event that one user becomes unresponsive in the state channel application, it is always possible to simply submit the latest state of the application to the `AppRegistry` contract. This requires submitting an `AppIdentity` object, the hash of the latest state, its corresponding nonce, a timeout parameter, and the signatures required for those three data (or, alternatively, a transaction where `msg.sender` is `owner` in the `AppIdentity`). This initiates a challenge and places an `AppChallenge` object in the storage of the contract assuming all of the information provided is adequate.

Counterparty is unresponsive *and* a valid action exists. In the case that an application adheres to the `AppDefinition` interface and provides valid `applyAction` and `getTurnTaker` functions, an action can additionally be taken when initiating a challenge. A function call to the `AppRegistry` with the latest state parameters exactly as in the case above but with an additional parameter for an encoded action and the requisite signatures by the valid turn taker can be made. In this case, a challenge is added the same as above but the state is progressed one step forward.

3.1.3 Responding to a Challenge

Cancelling the challenge. In the simplest case, you and your counterparty can always sign a piece of data that declares "we would both like to cancel this challenge and resume off-chain". In this case, simply provide the contract with this evidence and the challenge can be deleted and ignored.

Progressing the challenge on-chain. If the counterparty that initiated the challenge is not willing to continue the application off-chain, but their challenge is indeed a valid one (perhaps you were offline temporarily), then you can

progress the application on-chain if there exists an implementation for `applyAction` and `getTurnTaker`. In this case, much like is the case when initiating a challenge, you simply provide an encoded action and requisite signatures to re-issue the challenge but in the opposite direction.

Handling a malicious challenge. If the counterparty submitted stale state that is *provably malicious*, then the contract supports claiming this by submitting a provably newer state that convicts them and rewards the honest party (i.e., you in this case).

3.2 Resolutions

After a challenge has been finalized, the `AppRegistry` can now be used to arrive at a resolution. In the Counterfactual protocols, it is the *resolution* of an application that is important in executing the disribution of blockchain state fairly for any given off-chain application.

The resolution is defined in the framework as a `Transaction` structure. Note that this object is currently under consideration for a refactoring to be slightly more generalized (i.e., removing `assetType` and `token` to be specific), but as of today in the codebase it is represented by:

```
struct Transaction {
    uint8 assetType;
    address token;
    address[] to;
    uint256[] value;
    bytes[] data;
}
```

3.2.1 Setting a Resolution

After a challenge is finalized. If a challenge has been finalized by the timeout expiring, then a function call can be made to the `AppRegistry` that then initiates a call to the `resolve` function of the corresponding `AppDefinition` to the challenge. The `resolve` method will return a `Transfer.Transaction` struct and that is then stored inside the contract permanently as the resolution of the application.

In the same transaction as finalizing a challenge. A minor efficiency can be added here, but has not yet been implemented, which is that if the challenge can finalized unilaterally (either in initiation or in refutation) then it is possible to instantly set the resolution. There is an issue tracking this on GitHub.

3.3 FAQ

3.3.1 On-chain Progressions of Off-chain State

It is possible that an application may have a challenge initiated on-chain and then have some state updated correctly off-chain. This would likely only occur in the case of a software error, but nonetheless it is possible. In the case of a state machine progression then there is a uniquely non-fault-attributable scenario that can occur:

- Honest party A initiates a challenge on-chain with B after B is unresponsive at nonce `k`
- B comes back online and tries to update the state of the application by signing a unilateral action (thereby incrementing the nonce to `k + 1`) and sending it to A
- B *also* goes to chain and makes an on-chain challenge progression with a *different* action, thereby incrementing the `disputeNonce` to `1`

A is now in a bizarre spot where he can *either* respond to the challenge on-chain again (making the `disputeNonce` equal to `2`) or he could sign the state with nonce `k + 1`. In the latter case, the newly doubly-signed (`k + 1`)-versioned state would be able to be submitted on-chain and **overwrite** whatever state was on-chain with `disputeNonce` at `1`.

3.3.2 Provably Malicious Challenges

A challenge for an `n`-party off-chain application is considered to be provably malicious if the `nonce` of the challenge was `k` and someone was able to respond with a state signed by the `latestSubmitter` where the `nonce` of *that challenge response* was at least `k + 2`.

The reason why the same version of the above scenario with `nonce` equal to `k + 1` is not considered malicious is that the following situation might occur by an honest party:

- Honest party A signs state with nonce `k` and send it to B. Then, B countersigns and sends to A
- Honest party A signs state with nonce `k + 1` and send it to B, Then, B is unresponsive

In this situation, honest party A holds a signed copy of state with nonce `k` signed by B and can initiate a challenge on the blockchain. However, it is possible that B did in fact receive the signed state with nonce `k + 1` and can then respond to the challenge with this. Therefore, we must require that a state put on chain have at least `k + 2` to be considered an attempt at a stale state attack.

CHAPTER 4

Peer Protocol for Channel Management

To exemplify the protocols as we define them, we will assume there exists a multisignature wallet shared between two parties, Alice and Bob. This is the only required re-usable on-chain component (with the exception of supporting libraries) to execute each of the protocols below.

4.1 Primitive Types

4.2 Global Variables

4.3 JSON

This type specifies a modification of JSON that disallows the following primitive types: `true`, `false`, `null`. Note that when represented in javascript, large numbers which fint into `uint256` must be represented as either `BigNumber`s or as serialized `BigNumber`s (e.g., `{ _hex: '0x01'}`).

Type: `Terms`

Type: `CfAppInterface`

TODO: The name `getTurnTaker` needs to be standardized

CHAPTER 5

Setup Protocol

NOTE: All of the protocols below specify a 2-party interaction but can be generalized to the multi-party case in the future.

5.1 Messages

After authentication and initializing a connection, channel establishment may begin. All state channels must run the Setup Protocol before any other protocol. As the name suggests, its purpose is to setup the counterfactual state such that later protocols can be executed correctly.

Specifically, the Setup Protocol exchanges a commitment allowing a particular off-chain application to withdraw funds from the multisignature wallet. We call this application instance the Free Balance application, representating the available funds for any new application to be installed into the state channel. The app definition is called ETHBucket.

Unlike other protocols, there is no extra message data for the Setup Protocol because the commitment digests are fully determined by the addresses of the participants.

5.1.1 The `SetRootNonce` Message

5.1.2 The `SetRootNonceAck` Message

5.1.3 The `Setup` Message

5.1.4 The `SetupAck` Message

5.2 Commitments

Commitment for `SetRootNonce` and `SetRootNonceAck`:

The commitments that these two messages rely on have the following parameters:

The commitment can be visually represented like:

Commitment for `Setup` and `SetupAck`:

The commitments that these two messages rely on have the following explicit parameters:

Additionally, the following parameters are implicitly computed:

The commitment can be visually represented like:

> NOTE: The usage of `MultiSend` in this commitment is unnecessary and should be removed.

CHAPTER 6

Install Protocol

To illustrate the install protocol, first assume that the multisignature wallet owns 20 ETH and that the Free Balance application has recorded a balance of 10 ETH for both for Alice and Bob. Running the install protocol allows Alice and Bob to install an application where Alice and Bob both deposit 1 ETH to be disbursed based on the resolution logic of the application.

In this example, the application is Tic-Tac-Toe. You can see with the visual representation below that the funds available in the free balance decrease and the funds committed to the Tic-Tac-Toe application increase by the corresponding amount.

6.1 Messages

6.1.1 Types

First we introduce a new type which we label `InstallParams`.

Type: `InstallParams`

> NOTE: `signingKeys` are deterministically generated based on the nonce of the application in relation to the entire channel lifecycle. Further detail still to be provided in these specifications in the future. See this issue for discussion
>
> NOTE: At the moment, this message requires that the hexidecimal value of `peer1.address` is strictly less than the value of `peer2.address` to enforce deterministic ordering of the `signingKey` variable in new application installs. This can be improved in the future

6.1.2 The `Install` Message

6.1.3 The `InstallAck` Message

6.2 Commitments

Commitment for `Install` and `InstallAck`:

Let `c1` and `c2` be the amounts that parties 1 and 2 wish to contribute towards the application respectively. Then, the commitment should:

- Updates the state of the free balance application to one where the first party's balance is reduced by `c_1` and party the second party's balance should be reduced by `c_2`.
- Makes a delegatecall to `executeAppConditionalTransaction` with a limit of `c_1 + c_2` as also included in the terms.

The following parameters are included in the commitment:

The commitment can be visually represented like:

> NOTE: Although not shown in the visualization, the order of transactions is important. The `multiSend` must encode the call to `proxyCall` **before** the call to `executeAppConditionalTransaction`.

CHAPTER 7

Update Protocol

Once an application has been installed into the state channel, the multisignature wallet has transferred control over the installed amount from the free balance to the application's `resolve` function, a mapping from application state to funds distribution. For example, in the case of Tic-Tac-Toe, a possible payout function is: if X wins, Alice gets 2 ETH, else if O wins Bob gets 2 ETH, else send 1 ETH to Alice and Bob.

As the underlying state of the application changes, the result of the payout function changes. It is the job of the Update Protocol to mutate this state, independently of the rest of the counterfactual structure.

Using our Tic-Tac-Toe example, if Alice decides to place an X on the board, Alice would run the Update Protocol, transitioning our state to what is represented by the figure above. Notice how both the board changes and the *local* nonce for the app is bumped from 0 to 1. To play out the game, we can continuously run the update protocol, making one move at a time.

7.1 Messages

For the below messages, the digest that is signed is represented as the following:

```
keccak256(
  ["bytes1", "bytes32", "bytes32", "uint256",
  [
    0x19,
    keccak256(encode(
      [address, address[], address, bytes32, uint256 ],
      [owner, signingKeys, appDefinitionAddress, termsHash, defaultTimeout]
    )),
    0,
    TIMEOUT
  ]
);
```

Type: `UpdateParams`

7.1.1 The `SetState` Message

TODO: Add a field for the encoded action

7.1.2 The `SetStateAck` Message

7.2 Commitments

Commitment for `SetState` and `SetStateAck`:

The commitment can be visually represented like:

This transaction invoke the `setState` function with the signatures exchanged during the protocol.

CHAPTER 8

Uninstall Protocol

The lifecycle of an application completes when it reaches some type of end or "terminal" state, at which point both parties know the finalized distribution of funds in the application-specific state channel.

In the case of a regular application specific state channel, both parties might broadcast the application on chain, wait the timeout period, and then broadcast the execution of the Conditional Transfer, thereby paying out the funds on chain. In the generalized state channel context however, the post-application protocol is to transfer the funds controlled by the application back to the Free Balance application off chain, so that they could be reused for other off-chain applications.

Using our Tic-Tac-Toe example, imagine Alice made the final winning move, declaring X the winner. If Alice runs the Uninstall Protocol, then the Counterfactual state transitions to what is shown above.

8.1 Messages

8.1.1 The `Uninstall` Message

8.1.2 The `UninstallAck` Message

8.2 Commitments

Commitment for `Uninstall` and `UninstallAck`:

There are two key operations required for a successful uninstall.

- Set a new state on the Free Balance. The resolution function defined in the application must be run to compute an update to the Free Balance that is based on the outcome of the application.
- Set a new nonce on the Nonce Registry. As a result, the Conditional Transfer pointing at the original application will be invalidated and the application will be considered deleted.

Specifically, the Conditional Transfer commitment created by the Install Protocol checks that the dependency nonce does not equal 1. *If the nonce is ever 1*, then the conditional transfer will fail. Hence setting the nonce to 1 invalidates the conditional transfer, which is desired behaviour.

CHAPTER 9

Cleanup Protocol

> NOTE: Notice that the `stale-invalid state` object has been removed from the previous figure shown in the *Uninstall Protocol* representing the effective "garbage collection" phenomena of the cleanup protocol

The cleanup protocol is a protocol that is periodically run to update the dependency of every active application to a newer root nonce version. In effect, it achieves the goal of O(1) constant time invalidation of outdated state put on-chain by resetting the state such that regardless of the number of historical / outdated apps that have been uninstalled, refuting their valididity on chain requires a single transaction.

9.1 Messages

> NOTE: The dependency in the message exchange is important; it is not safe to sign the root nonce commitment without possession of all the active app commitments.

9.1.1 The `CleanupInstall` Message

9.1.2 The `CleanupInstallAck` Message

9.2 Commitments

Commitments for `CleanupInstall` and `CleanupInstallAck`:

For each active application, a similar commitment to the one described in the *Install Protocol* must be generated. The commitment calls `executeAppConditionalTransaction` with a limit of `c_1 + c_2` and a expected root nonce key of `r + 1`. Note that this is different from the install commitment in that it is not a multisend and does not set the free balance. Note that the free balance is also considered an active app. Here is an example of a commitment for a given app:

Then, finally, the commitment update the root nonce is simply:

CHAPTER 10

Install Virtual App Protocol

This is the Install Virtual App Protocol.

10.1 Roles

Three users run the protocol. They are designated as `initiating`, `responding`, and `intermediary`. The first two parties are the users wishing to interact together in a virtual app, who do not necessarily have a ledger channel between them. It is required that `initiating` and `intermediary` have a ledger channel together, and that `responding` and `intermediary` have a ledger channel together.

10.2 The `InstallVirtualAppParams` type

10.3 Derived fields

These fields are not included in `InstallVirtualAppParams` but are computed from existing information known to a user.

`{initiating,responding,intermediary}`, together with the target app sequence number, are used to derive the app-specific signing keys. `signingKeys[0]` is the intermediary signing key, while `signingKeys[1:2]` are the signing keys used by `initiating` and `responding`, sorted lexicographically by public key.

10.4 Commitments

10.4.1 leftETHVirtualAppAgreement

A commitment to call `ETHVirtualAppAgreement::delegateTarget` with an `Agreement` argument with the following fields

10.4.2 rightETHVirtualAppAgreement

A commitment to call `ETHVirtualAppAgreement::delegateTarget`

terms:

`limit` is explicitly ignored by the contract, while `assetType, token == 0, 0` means ETH.

10.4.3 targetVirtualAppSetState

The protocol produces a commitment to call `virtualAppSetState` with the initial state. Note that `intermediary` produces a "type 2" signature while the others produce a "type 1" signature. This ensures that the intermediary's signature can be reused for calling `virtualAppSetState` with other app state hash values, i.e., that the intermediary does not need to be part of the update-virtual-app protocol.

```
d1 = keccak256(
  ["bytes1", "bytes32", "bytes32", "uint256",
  [
    0x19,
    keccak256(identity),
    0,
    TIMEOUT
  ]
);
```

```
d2 = keccak256(
  ["bytes1", "bytes32", "uint256", "uint256", "bytes1"],
  [
    0x19,
    keccak256(identity),
    65536,
    TIMEOUT,
    byte(0x01)
);
```

The signatures `s5` and `s7` are the signatures of `initiating` and `responding` respectively on `d1` while the signature `s6` is the signature of `intermediary` on `d2`.

10.5 Messages

10.5.1 M1

10.5.2 M2

10.5.3 M3

10.5.4 M5

10.5.5 Summary

CHAPTER 11

Uninstall Virtual App Protocol

This is the Uninstall Virtual App Protocol.

11.1 Roles

Three users run the protocol. They are designated as `initiating`, `responding`, and `intermediary`. It is required that `initiating` and `responding` have run the `install-virtual-app` protocol previously with the same `intermediary`; however it is allowed to swap the roles of `initiating` and `responding`.

11.2 The `UninstallVirtualAppParams` type

At the end of this protocol the commitments `{left,right}ETHVirtualAppAgreement` defined in the `install-virtual-app` protocol are cancelled, and the free balances are updated.

11.3 Commitments

11.3.1 lockCommitment

The protocol produces a commitment to call `virtualAppSetState` with the final state of the app and at a high app local nonce. The existence of this commitment and the high local nonce means that the `s6` signature (from `intermediary` on the `targetVirtualAppSetState` commitment) is no longer useful, and the changes to the app state cannot be made without the intermediary's signature.

11.3.2 uninstallLeft

A commitment to cancel the `leftETHVirtualAppAgreement` commitment produced by `install-virtual-app` and simultaneously update the free balance in the `initiating-intermediary` free

balance.

11.3.3 uninstallRight

A commitment to cancel the `rightETHVirtualAppAgreement` commitment produced by `install-virtual-app` and simultaneously update the free balance in the `intermediary-responding` free balance.

11.4 Signatures

11.5 Messages

11.5.1 M1

11.5.2 M2

11.5.3 M3

11.5.4 M4

11.5.5 M5

11.5.6 M6

11.5.7 M7

11.5.8 M8

CHAPTER 12

Withdraw Protocol

12.1 The `WithdrawParams` type

12.2 Commitments

12.2.1 installRefundApp

This is exactly the same kind of install commitment produced by the install protocol for regular apps. This commitment installs a balance refund app.

12.2.2 withdrawCommitment

This is a commitment for the multisig to send `amount` wei to `recipient`.

12.2.3 uninstallRefundApp

This is exactly the same kind of uninstall commitment produced by the uninstall protocol for regular apps and uninstalls the app installed by `installRefundApp`.

12.3 Signatures

12.4 Messages

CHAPTER 13

Glossary and Terminology Guide

For an introduction to the concepts and terminology behind state channels, please see the original paper.

13.1 State Deposit

Any kind of blockchain state controlled directly by a state channel. This could be an ETH balance, ownership of an ERC20 token, control over an ENS name registration, or any other kind of state.

13.2 State Deposit Holder

The on-chain multisignature wallet smart contract that is the "owner" of a given state deposit

13.3 Counterfactual Instantiation

The process by which parties in a state channel agree to be bound by the terms of some off-chain contract

13.4 Counterfactual Address

An identifier of a counterfactually instantiated contract, which can be deterministically computed from the code and the channel in which the contract is instantiated

13.5 Commitment

A signed transaction (piece of data) that allows the owner to perform a certain action

13.6 Action

A type of commitment; an action specifies a subset of transactions from the set of all possible transactions conditional transfer: the action of transferring part of the state deposit to a given address if a certain condition is true.

> NOTE: Section 6 of the paper specifies a concrete implementation that differs in certain respects from the protocol described here. The reason for this divergence is explained later.

CHAPTER 14

Contributing

HTML files are built automatically by readthedocs. To build a local preview copy, a Makefile is provided that invokes sphinx. Install the dependencies and build a local preview by doing the following:

```
python3 -m venv venv
source venv/bin/activate
pip3 install -r requirements.txt
make html
```