Optimistic Game Semantics

Vision for a Unified Layer 2

Ben Jones, Karl Floersch

January 2020

Abstract

The proliferation of blockchain layer 2 (L2) advancements has created a complex design landscape with minimal interoperability between L2 clients and applications. In reality, there are large overlaps in architectural design inherent to many L2s, and we believe it would benefit the ecosystem for L2s to share infrastructure, security proofs, and logical models rather than reinventing the wheel each time.

This work introduces a first step towards a unified L2 theory: Optimistic Game Semantics. These semantics interpret logical expressions as dispute-based game trees, and evaluates truth as the existence of a winning strategy. A variety of L2 protocols can be expressed under this framework–in this paper, we demonstrate that state channels, plasma, and cross-shard state schemes are all logical expressions in the semantics. Further, a single smart contract can adjudicate disputes for them all.

We believe that this motivates the creation of a generic, shared infrastructure capable of executing many L2 protocols.

1 Introduction

1.1 Motivation

Over the past several years, the blockchain layer 2 (L2) space has rapidly expanded. From virtual [1] and generalized [2] state channels, to dozens of plasma and commit-chain constructions, a complex design landscape has begun to emerge. Even L1 research into cross-shard communication has led to L2-like patterns. [3]

Despite an increasing number of designs, the core idea remains the same: optimistic execution. All L2 protocols take in the current state of the L1 blockchain, along with some local information and assumptions outside of L1, and output a prediction about future L1 state. Further, research has demonstrated many compositions of L2 protocols: channels on channels, channels on commit-chains, and other hybrid approaches have emerged.

We believe that diversity in the L2 design space is an opportunity for abstraction, on which secure, interoperable infrastructure can be built. The future will see many designs in use, not "one L2 to rule them all."

1.2 Vision

We envision a general-purpose L2 node which can run many L2 protocols under a unified architecture. Particularly, all L2 clients seem to implement two fundamental components:

1. **Off-chain State Solver**: Given the current L1 state, along with some off-chain data and assumptions, make predictions about future L1 state.
2. **Off-chain State Transitioner**: Given user input, execute the rules defined by the L2 protocol to change the predicted L1 state.

An example of a State Solver in use is a channel client tracking a balance. Based on L1 deposits into the channel contract, and the channel's off-chain messages, the client determines "state" of the channel: "I can withdraw 2 coins from this contract in the future."

In comparison, the transitioner is the part of the channel client that sends money at the user's request, e.g. by signing a new balance and sending it to their counter-party. This represents a "state transition" for the client: "send 1 coin in this channel to Bob, so that he can withdraw 1 more coin in the future."

With the right abstraction, bespoke L2 implementations could be replaced with shared infrastructure–that is, an L2 client whose State Solver and Transitioner are generalized enough to operate a variety of L2 protocols, instead of just one.

1.3 Our Contributions

While a general-purpose L2 client sounds useful, how to actually build it has been unclear: L2 research usually involves one-off designs with tightly coupled implementations, lacking a unified theory which would afford a generic implementation. In this work, we present the first step towards a unifying L2 theory.

Particularly, we demonstrate a new model for L2 constructions: Optimistic Game Semantics. Many L2 solutions, including state channels and plasma, correspond to particular expressions within these semantics. Further, a single smart contract can use the semantics to adjudicate the entire family of L2 solutions, and a single client can interpret off-chain state for the entire family.

Syntactically, the language is built on predicate logic. Our work introduces a new interpretation of the logic, based on game semantics, which imagines each logical expression as a two-player game tree. "Truth" in the semantics corresponds to the existence of a winning strategy for one of the two players, and the true expressions represent the L2 state.

Further, the semantics provide a shared lens through which to view L2 designs. Since the game trees provide a comprehensive view of the dispute structure, they semantics provide a rigorous foundation on which to build security proofs. This is especially an improvement for plasma, where safety proofs were bespoke and non-rigorous.

We believe this work suggests that a generalized L2 client is viable in practice.

2 Optimistic Game Semantics

In this section, we will introduce the semantics of L2 games. This involves two key aspects:

- The interpretation of logical expressions as game trees describing on-chain disputes.
- The interpretation of truth of these expressions as game strategies, based on the computability assumptions of cryptographic primitives.

Intuitively, the first aspect might say "if Alice claims that $A \wedge B$, Bob can dispute by claiming either $\neg A$ or $\neg B$." The second aspect says "since Alice can prove each of A and B, she will win."

Together, these components interpret logical expressions as L2 constructions. This means that L2 constructions can be created simply by writing new logical expressions, as opposed to implementing custom on-chain dispute logic and off-chain state interpreters.

2.1 Gameplay Setup

We start by constructing a game semantic interpretation of logical expressions. We will use a language L based on the usual first-order syntax; that is, using the propositional connectives $\wedge, \neg, \vee$ and the quantifiers $\forall, \exists$. For convenience, we will use the following notation for generic expressions in L:

- p, p_0, and p_1 will be any proposition (closed, well-formed formula) in L.
- $p(x)$ will be any well-formed formula in L in which x is free.
- $p(x/t)$ will be the well-formed formula in L in which all instances of the free variable x in $p(x)$ are bound to t.

A "challenge game" will be played between two players: a prover P and an opponent O. We will use A and B as variables for these two players, where $A \neq B$.

All game states are of the form of A, p: that is, the current game state is a "current proposition" p tupled with a "defending player" A. We say some game state A, p is "A-signed."

The game progresses according to the current state and a set of "challenge rules". The challenge rules are effectively a turn-based generalization of the challenge-response protocols commonly used in L2 constructions. The challenge rules are as follows:

Game State	Challenge Input	New Game State
$A, \neg p$	$\emptyset$	B, p
$A, p_0 \wedge p_1$	$i \in \{0, 1\}$	$B, \neg p_i$
$A, p_0 \vee p_1$	$\emptyset$	$B, \neg p_0 \wedge \neg p_1$
$A, \exists x \in X : p(x)$	$\emptyset$	$B, \forall x \in X : \neg p(x)$
$A, \forall x \in X : p(x)$	$t \in X$	$B, \neg p(x/t)$

Challenge rules for the logical connectives and quantifiers.

Observe that not all challenges allow a choice on behalf of the challenging player; e.g. the game state of the form $(P, \neg p_0)$ has only one game state immediately proceeding: (O, p_0).

2.2 Atomic Predicates and Game Outcomes

After a series of challenges, the game state will eventually reduce to some closed, atomic formula in L–for example, $x \geq y$ or $\texttt{hash}(h_0) = H_0$. We will denote these predicates as $a, b, ...$ and assign them Boolean truth values. Since atomic formulas have no strict subformulas, they have no further corresponding challenges. Thus, they correspond to leaves of the game tree, and must be used to assign game outcomes.

We define two Boolean game outcomes, corresponding to a win by P or O. The game outcome for a terminal game state A, p is determined as follows:

- If p is some atomic predicate a with $a = true$, the game is won by A.
- If p is some atomic predicate a with $a = false$, the game is won by B.
- If p is not an atomic predicate, the game is won by A.

In other words, a player wins if the game terminates with them stating a true atomic proposition, or if it terminates with their opponent stating a false atomic proposition. The third condition covers the case where a player does not challenge–for example, $\forall x \in \emptyset : a(x)$ is won by P since O cannot play an element of the empty set ("vacuous truth").

2.3 Game Tree and Strategies

With the above definitions, we can fully define the game tree for a given proposition p. Specifically, it is a tree with the root P, p (the initial game state), vertices for all possible future game states based on the challenge rules, and branches/edges between each game state and its possible challenges. For example, the game tree for $F \vee T$ (where F and T are respectively false and true atomic formulas) is:

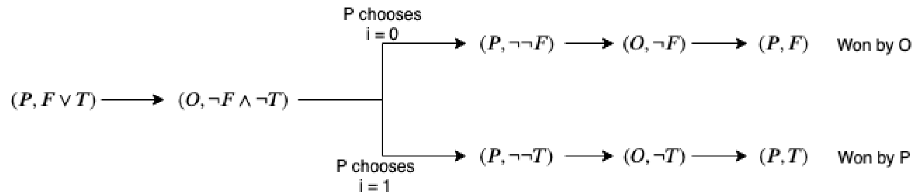

Figure 1: Game tree for $T \vee F$.

A "strategy" for the player A for a particular proposition p is a subtree S of the game tree with the following properties:

- The root P, p is contained in S.
- For each A-signed game state in S, all corresponding B-signed challenges are contained in S.
- For each B-signed game state in S, there is exactly one A-signed child state.

In other words, a strategy for player A enumerates a unique response to all possible moves by B until the game has ended. For example, the following subtree would be a strategy for P playing the previously specified game $T \vee F$ (highlighted in green):

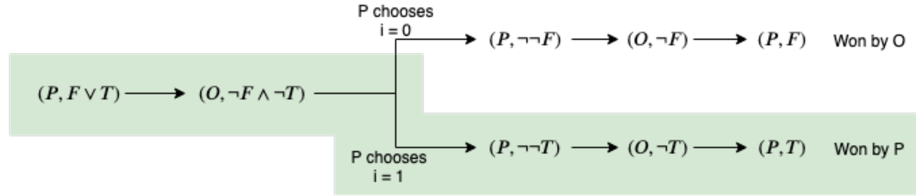

Figure 2: A winning strategy for P playing $T \vee F$. (in green)

If there exists a strategy whose leaves all result in a win for player A, we say that game has a winning strategy for A. In the Optimistic Game Semantics, we interpret a proposition as true if its corresponding game has a winning strategy for the prover P, and false if there is a winning strategy for the opponent O.

Observe that a proposition will always be either true or false, and never both. Further, if A has a winning strategy for p, then B has a winning strategy for $\neg p$: the $\neg$ connective "switches the roles" of P and O, so that if p is true, $\neg p$ is false, and vice versa.

2.4 Extending to Cryptography: Computability Models

This above construction is very similar to existing game semantics, especially dialogical logic. Historically, these were constructed to match some existing system of computation–here, we want to use them to build L2 protocols. To do this, the model must incorporate cryptography. Particularly, moves in the game tree are restricted to those which are actually computable by an L2 agent.

For example, imagine the atomic predicate $\texttt{is_hash_preimage}(h, H)$, which uses a cryptographic hash to evaluate whether $\texttt{hash}(h) = H$. The playable moves in the propositional game g

$$g := \exists h \in \texttt{bytes} : \texttt{is_hash_preimage}(h, H_0)$$

should be dependent on whether P knows such a preimage h_0.

We will call the additional structure which accounts for this a "computability model." A computability model consists of two components:

- A set W of "worlds" parameterizing the computability of moves in the game. For example, one $w \in W$ might represent that "the preimage h_0 is secret to P, and the preimage h_1 is known by both P and Q."
- For every set X which appears in $\forall$ or $\exists$, a function X_{comp}:

  $$X_{comp} : W \times \{P, O\} \rightarrow \mathcal{P}(X)$$

  Where $\mathcal{P}$ is the powerset function, so that $X_{comp}(w, A) \subseteq X$. Effectively, $X_{comp}(w, A)$ is the "computable subset" of X accessible to player A in world w.

With the introduction of computability models, truth of a proposition p is no longer meaningful "in a vacuum." This is because the game tree, and thus the existence of a winning strategy for either P or O, is dependent on w. Specifically, to evaluate truth "in world w," we alter the challenge rule for the universal quantifier:

Game State	Challenge Input	New Game State
$A, \forall x \in X : p(x)$	$t \in X_{comp}(w, B)$	$B, \neg p(x/t)$

Challenge rule for universal quantification with computability restrictions.

Thus, the game tree is dependent on the world w. If the tree for some p in world w has a winning strategy for P or O, we say p is true or false "in w."

2.5 Minimal Computability Model

Since the focus of this work is on the games, we will construct a maximally simple computability model to be used throughout the rest of this paper.

First, we will consider "idealized" cryptographic predicates. Particularly, we will ignore collisions, so that h_0 is considered the unique input for which `is_hash_preimage` returns true against its image H_0. Similarly, assume a `verify_signature` such that signatures are unique to a given message and public key, and a Merkle `verify_inclusion` for which proofs and leaf values are unique to a given root and leaf key.

Next, we will examine the set of worlds W as

$$W : I_{pub} \times I^P_{sec} \times I^O_{sec}$$

Where each $i_{pub} \in I_{pub}$ denotes a set of "public information", and each $i^A_{sec} \in I^A_{sec}$ denotes a set of "information secret to A." Any $w \in W, w = (i_{pub}, i^P_{sec}, i^O_{sec})$ will always hold the invariant that secret information is not public, so that

$$x \in i^A_{sec} \Rightarrow x \notin i_{pub}$$

always holds true. Then, for a given world $w = (i_{pub}, i^P_{sec}, i^O_{sec})$, we define the computable moves for either player as

$$X^A_{comp}(w, X) := X \cap i_{pub}$$

In other words, a player can only challenge universal quantification with moves from the public information set i_{pub}.

For example, here is a winning strategy for the hash preimage game g if the world w has $h_0 \in i_{pub}$:

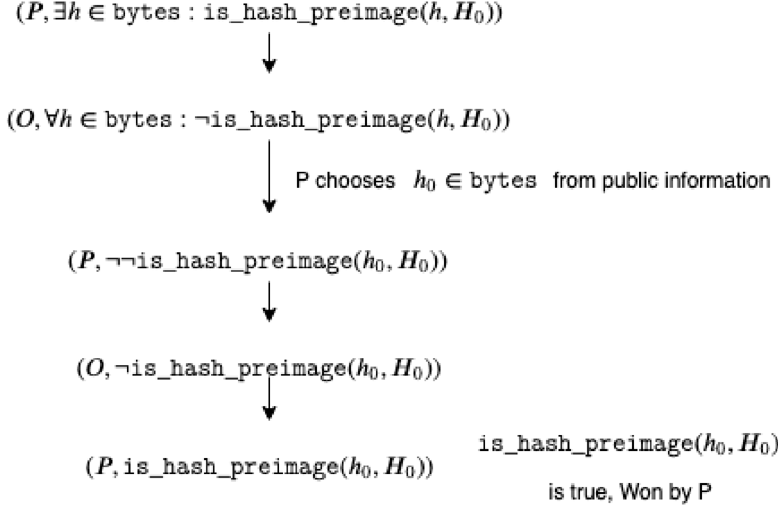

Figure 3: A winning strategy for P playing g if they know h_0.

So, g is true if the h_0 is public. However, if instead $h_0 \in i^O_{sec}$, it cannot be played, so the game is winning for O, and thus false. (see Figure 4)

A more rigorous, Universal Composability-style treatment of the computability model is beyond the scope of this work, but an important area of future research. Nonetheless, it is sufficient for our purposes of describe L2 protocols. Protocol execution–that is, the passing of messages, signatures, etc. between the players–represent progressions through different worlds.

2.6 Example: Preimage State Channel

At this point, we can construct a game which corresponds to a primitive unidirectional state channel! In a unidirectional state channel, a sending party locks funds in a contract which pays out to a recipient if they can provide some information initially known only by the sender. Thus, by revealing the secret information, the sender can make off-chain payments to the recipient.

Imagine a set of indexed hash preimages $h_0, h_1, ..., h_n$ and corresponding hash images $H_0, H_1, ..., H_n$. Now consider the games defined by

$$channel(s) := \forall i \in \{s+1, s+2, ..., n\} : \neg\exists h \in \texttt{bytes} : \texttt{is_hash_preimage}(h, H_i)$$

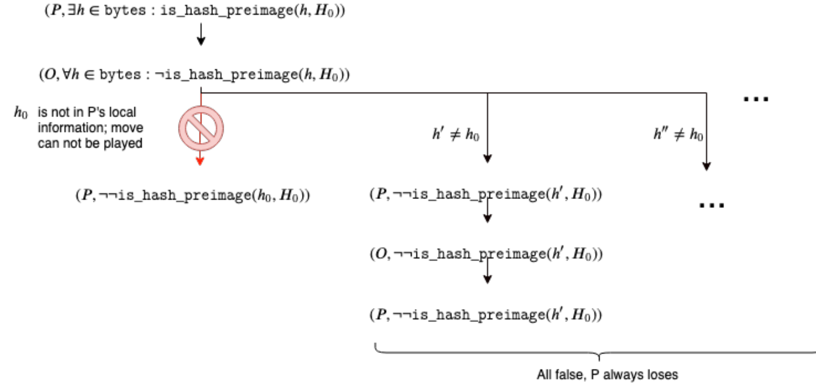

Figure 4: A losing tree for P playing g if they do not know h_0.

Imagine an initial world w_0, in which every $h_i \in i_{sec}^P$–in other words, all the preimages are secret to P. In this world, P is able to win all possible games of $channel(s)$, so $channel(0), channel(1), ..., channel(n)$ are all true. For example, Figure 5 shows P's winning strategy for $channel(0)$.

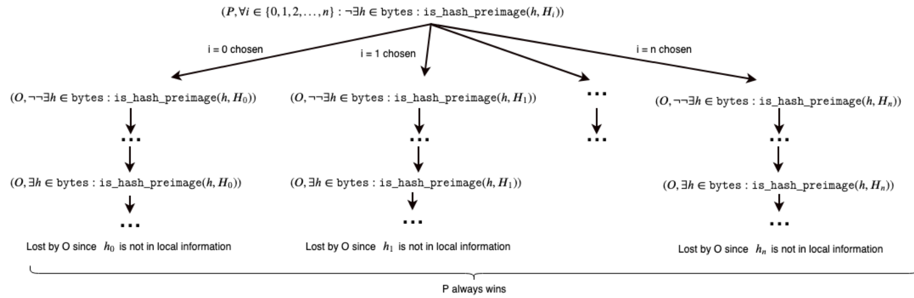

Figure 5: Winning strategy for P playing $channel(0)$ if O does not know any h_i.

Now, imagine that P and O execute a message passing protocol whereby P sequentially reveals the hash preimages. This results in a progression of computability worlds, $w_1, w_2, w_3, ...$ so that w_j has $i_{pub} = \{h_0, h_1, ..., h_{j-1}\}$. at this state, $channel(0), channel(1), ..., channel(j-1)$ no longer have winning strategies for P, and are thus "false."

3 Game Contract Implementation

While it is interesting to describe L2 constructions in a shared language, we ultimately need to actually build them. This section shows how to utilize the

semantics in practice. a Particularly, we can deploy a single, generic smart contract on a stateful blockchain (e.g. Ethereum) which can instantiate games satisfying the above semantics. This enables layer 2 constructions to be implemented simply by defining them as logical expressions in L.

The primary components are to assign cryptocurrency payouts based on game outcomes, so that the L2 protocols are useful, and to design a contract architecture which works in a permissionless, multiparty setting.

3.1 Contractual Payouts

To realize the above example game as a state channel, we need to describe how cryptocurrency is distributed based on game outcomes. Imagine that P deposits N coins into a smart contract which can instantiate games of the above form: at any time, P may specify an s, beginning a game of $channel(s)$. If the game is won by P, the contract pays out $N - s$ coins to O and s coins to P.

Thus, by sequentially progressing each w_i, P is able can make a series of 1 coin payments to O. O is confident in their receipt of funds, not because anything has happened on chain, but because of their ability to play the uniquely winning strategy, and because doing so will give them the money.

We call the generic implementation of this functionality a "deposit contract." It is specified by some $q(s)$, an open formula in L, and a payout function accepting the same input, $p(s)$. If a game of $q(s/t)$ is ever won, then the deposit contract executes the function $p(t)$.

3.2 Permissionless Contract with Multiple Parties

3.2.1 Intiution for the Permissionless Context

If we want to create a blockchain dispute contract which evaluates these games, we need to open the participant set: in reality, L2 protocols often involve more than two participants. Thus, we need games which facilitate interactions between more than just two individuals P and O.

Luckily, we can construct a permissionless ("anyone can make a move") game which has the exact same strategies as the two-player model above. To achieve this, we allow "plays" of the game to encompass multiple branches of the game tree, instead of singe progressions of game state. This is not to be interpreted as a "new game" with states progressing as $(A, p_0), (B, p_1), (C, p_2), ...$ and so on. Instead, we can think of players $A, B, C, ...$ as playing individual moves "on behalf" of P and O in the above semantics. The contract is constructed so that, if any player sees a winning strategy for P or O, they can force the overall game to be won or lost regardless of what any other player does.

3.2.2 Adjudication Contract

Imagine an "adjudication contract" C_{adj} which maintains a directed graph of games, where a game g is defined by the following values:

- `prop` - the proposition p in the langage L the game corresponds to.
- `created_block` - the `block.number` in the blockchain at which the game was instantiated.
- `challenges` - a list of the other games in C_{adj} challenging this one.
- `decision` $\in$ {`true`, `false`, `undecided`} - the current status of the adjudication contract's decision or this game.

Assume that there is some "dispute period" of T blocks. C_{adj} allows the following state transitions:

- Any account on the blockchain can make a call to a function C_{adj}.`instantiate_game`(p), instantiating a new game. That game g will be given:
  - g.`prop` $= p$
  - g.`created_block` = `block.number`
  - g.`challenges` $= \emptyset$ (no initial challenges)
  - g.`decision` = `undecided`
- Any account can call C_{adj}.`add_challenge`(g_0, g_1). If the following conditions are met:
  - g_1.`prop` is a valid challenge to g_0.`prop` according to the challenge rules.
  - g_0.`decision` $\neq$ `true`, i.e. g_0 has not already been decided true.

  Then g_1 is added to g_0.`challenges`.
- If g.`prop` is some atomic proposition a, then an authenticated contract implementing the functionality of a can make a call to C_{adj}.`set_atomic_decision`(g, *bool*), setting g.`decision` to either `true` or `false`.
- If for any g_0, g_1 such that g_1.`decision` = `false` and $g_1 \in g_0$.`challenges`, a call to C_{adj}.`remove_challenge`(g_0, g_1) will remove g_1 from g_0.`challenges`.
- If for any g_0, g_1 such that g_1.`decision` = `true` and $g_1 \in g_0$.`challenges`, anyone can call C_{adj}.`falsify_from_true_challenge`(g_0, g_1), which will set g_0.`outcome` = `false`.
- At any g.`created_block`+$T <$ `block_number`, users can call C_{adj}.`make_decision`(g). If f g.`challenges` is empty, the game is decided g.`outcome` = `true`.

3.2.3 Strategy Equivalence for 2-player Semantics and C_{adj}

The game states in C_{adj} are now subtrees of the game tree, as opposed to individual nodes like in the 2-party construction. However, the existence of a winning strategy for p in the 2-player game will always correspond to winning the equivalent game in C_{adj}.[1] Recall the definition of a winning strategy S for the two-player game: for each P-signed game state in S, all corresponding O-signed challenges were contained in S, and for each O-signed game state in S, there is exactly one P-signed child state, which is winning.

Winning a game in this multi-party setting requires one simple modification: instead of responding to one particular challenge brought forth by O, players respond to all such challenges brought forth by any participant. Since the strategy already enumerated a response for all such challenges, playing multiple runs of the game in parallel will not affect the ability to force a game outcome– as long as the correct challenge is one of the ones played, the contract outcome will mirror the 2-player outcome. An illustration of this parallel is demonstrated below:

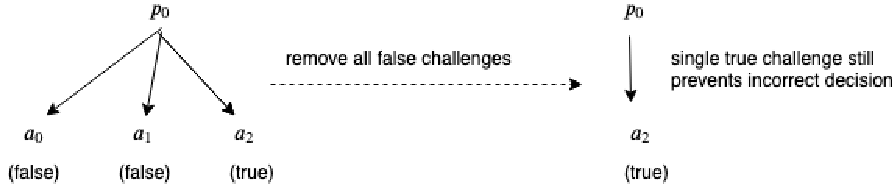

Figure 6: Demonstration of why the multiparty contract mirrors the strategies from the two party case.

Thus, we have created a smart contract which allows any account on the blockchain to permissionlessly make moves, while ensuring that strategies in the two-party semantics hold true for the n-party contracts. Note that now, P and O are "roles that may be played," rather than specific actors. Some clients $A, B, C, ...$ can be thought of as actors that might play either of these roles at various times.

3.3 Impact on Computability and Protocol Execution

While the same strategies apply for this new game architecture, the multi-agent setting does introduce more complexity for the computability model. Specifically, different clients will have downloaded particular subsets of the public information. A simple way to handle this is, instead of just one i^P_{pub}, have $i^A_{local}, i^B_{local}, i^C_{local}, ...$, downloaded subsets of the local information for each client in the system.

[1] At least, for the examples we will cover below. Technically, the contract design above does introduce some edge cases in which the game can become a DAG instead of a tree, for example games of the form $p \vee \neg p$. These will be ignored for the purposes of this work.

This will introduce cases in which a client A cannot determine whether there is a winning or losing strategy for some proposition p. However, this just means that the client can track a subset of the global state–the part relevant to its vested interest according to the protocol. Well-defined protocols should always enable the client to store the sufficient information to recover funds for that particular protocol.

4 Examples and L2 Classification

Let's look at how a variety of L2 constructions can be implemented using the game semantics and C_{adj}. Defining the L2 protocol involves defining two things: the game and payouts of the deposit contract C_{dep}, and the message-passing protocol which agents execute to progress the state through worlds in W.

4.1 State channels

4.1.1 Game Construction

While we demonstrated a very primitive channel construction in the two-party context, we can provide a better construction which mirrors Ethereum-style channels. Consider a set of participants `participants` and a data structure `state` which specifies an integer `state.nonce` and a mapping `balances` which assignes each of the `participants` an integer value.

The deposit contract C_{dep} custodies some coins on behalf of some specified `participants`, and a payout $p($`state`$)$ which sends coins amounting to the `state.balances` to each of the `participants`.

For convenience, we will define a multisig helper predicate:

$$\texttt{multisig_exists}(\texttt{participants}, \texttt{state} \in \texttt{STATES}) := \\ \forall p \in \texttt{participants} : \exists sig \in S : \texttt{verify_signature}(\texttt{state}, p, sig)$$

The game constructor $g($`state`$)$ for C_{dep} can now be defined, as follows:

$$\text{g}(\texttt{state}) := \texttt{multisig_exists}(\texttt{participants}, \texttt{state}) \\ \wedge \neg \exists \texttt{higher_state} \in \texttt{STATES} : \\ \texttt{multisig_exists}(\texttt{participants}, \texttt{higher_state}) \wedge \texttt{higher_state.nonce} > n$$

In other words, "I claim that this `state` was signed by all participants, and that there does not exist one with a greater nonce signed by all participants."

4.1.2 Protocol Execution

The participants can follow conventional state channel protocols to update the winning state. By signing messages with a monotonically increasing nonce, they are able to reallocate balances between themselves. Note that, as is consistent with conventional state channel wisdom, only the `participants` will be able to ensure that they have a winning strategy for the current state, because only

they know that their private key is secret and thus there exist no higher states with a valid multisig. This is why state channels have a bounded participant set.

4.2 Optimsitic Rollup

One useful computability primitive which has not been discussed so far is the blockchain itself. Particularly, because blockchains are censorship resistant, they can enforce that a set of data is available because it is a part of the L1 chain, e.g. using the "rollup" mechanism invented for Ethereum zk-Rollup.[4] This enables a computability model in which $X_{comp} = X$, making strategies very easy to calculate and reason about.

4.2.1 Game Construction

Imagine an authenticated log of available data on the blockchain, to which an agent called the aggregator can append tuples of the form $(transition, state)$. We will call the ith such value $(transition_i, state_i)$. Further, assume some atomic `verify_state_transition`$(preState, transition, postState)$ predicate which verifies transitions for a state machine, for example the EVM. If in world w, N such transitions have been appended to the log, then the game:

$$\text{valid_rollup_chain(N)} := \forall i \in \{1, 2, ..., N\} : \\ \texttt{verify_state_transition}(\text{state}_{i-1}, transition_i, state_i)$$

Allows for disputes which act as a fork choice for the rollup chain. The exact payout scheme depends on `execute_state_transition`, but the general idea is that a type of state transition which "burns" rollup assets should allow for them to be withdrawn back to L1.

4.2.2 Protocol Execution

Assuming `execute_state_transition` implements a signing-based transfer function, then users can sign transactions, and send them to the aggregator. Once these are "rolled up" (appended to the on-chain log), the client will update the user's balance.

Note that there are many extensions to this design. One important one is the ability to roll up only the *transition*, along with a 32-byte *stateRoot*, as the transitions should be significantly smaller than the total state. Merkle-based fraud proof predicates then enable disputes to be logarithmic in the state size. Another extension is to add more complex quantification to the game, so that a branching fork choice is possible.

4.3 Plasma

4.3.1 Game Construction

The various plasma-like designs, such as Plasma Cash, NOCUST, etc... represent a family of L2 constructions which utilize a "commit-chain." Imagine that some party, which we will call the aggregator, is given the ability to append to a log of 32 byte hashes. We will refer to these hashes as $r_0, r_1, ...$, and they will be the roots of an accumulator such as a Merkle tree.

Using such a commit-chain, we will construct a simple Plasma Cash variant, Plasma Lite.[2]

We will use the atomic predicate $\texttt{verify_inclusion}(value, index, proof, root)$ which verifies inclusion of a *value* at the given leaf *index* in a Merkle tree with the given *root* via an inclusion proof, *proof*. Assume that each of the *value*s included is a `state_update` $\in$ `S_U` of the form (`owner, sending_key_image`), where the `owner` is a public key and the `sending_key_image` is a 32 byte hash image.

The game constructor for the deposit contract is:

$$\begin{gathered}
\text{plasmaExit(coin, block, } \texttt{state_update}\text{, proof):=} \\
\texttt{verify_inclusion}(\texttt{state_update}\text{, coin, proof, } \mathrm{r}_{block}) \\
\wedge(\neg\exists preimage : \\
\texttt{is_hash_preimage}(preimage, \texttt{state_update.sending_key_image})) \\
\wedge\forall b < block : \forall \texttt{old_state_update} \in \texttt{S_U} : \\
(\ \neg\exists proof : \texttt{verify_inclusion}(\texttt{old_state_update}, c, p, r_b) \\
\vee\exists preimage : \\
\texttt{is_hash_preimage}(preimage, \texttt{old_state_update.sending_key_image}\ ))
\end{gathered}$$

Basically, the *plasmaExit* game states the following: "I claim that the preimage for this `state_update`, which was included at this block for this coin, has not been revealed, and that all `old_state_update`s included at previous blocks for this coin have been revealed.

The payout function $p(\texttt{state_update}, coin, proof, r_{block})$ pays 1 coin on-chain to the `state_update.owner`.

4.3.2 Protocol Execution

To successfully send a coin off-chain in this protocol, that the current the aggregator must first included a new `state_update`, specifying the current owner's desired recipient and a hash image to which only the recipient knows a preimage. Once the current owner has verified the inclusion of such a `state_update` in a future root r of the commit-chain, they publicize their `sending_key_image` to the recipient to complete the off-chain transaction.

[2]Special thanks to Dan Robinson for this construction.

5 Conclusion

5.1 Future Work

5.1.1 Extensions to the Semantics

The Optimistic Game Semantics presented above could afford a number of additional connectives. One example of this from the plasma literature is Plasma MVP's "exit queue," which determines validity based on a summation operation over an ordered list of subgames. Another example are the binary search games like those used in Truebit[5]. This game can be expressed using the logical connectives above, but the resulting expression is unwieldy.

One other nuance of the semantics which have been largely ignored for this paper is challenges which require no input $(\neg, \vee, \exists)$. In practice, C_{adj} should not waste an entire $2T$ blocks when evaluating the expression $\neg\neg p$. This example corresponds to a broader class of extensions which force certain subgames to be played up front, for example forcing a user to prove some predicate "up front" when exiting a plasma state.

5.1.2 Protocol Transitioner

The biggest open challenge to defining a generalized L2 client end-to-end is the "State Transitioner," that is, the part of the client which accepts user input and performs the corresponding protocol execution. The transitioner would presumably encompass a more rigorous computability model, and specify a way to define when to perform actions like message signing and sending based on a protocol definition. Together, these components would provide an end-to-end safety proof for L2 protocols–a historically elusive target, for non-channel L2s .

One possible approach to this analysis is Kripke semantics, a formal semantics for expressing the modal logic operators "necessarily" and "possibly". Our concept of the "possible worlds" W was in fact borrowed from Kripke semantics, which also specify a binary "accesibility relation" between elements of W. This accessibility might be able to capture L2 protocol execution, as other distributed systems models have been built with Kripke semantics.

5.2 Implications of this Work

We believe that, with the maturation of the L2 space, unification and shared infrastructure will be key to success. As L2 continues to diversify, it seems increasingly likely that no "one L2 to rule them all" exists–rather, different L2 applications are suited for different use cases. Because individual users will want to employ many of these use cases, it becomes critical that the applications are interoperable. This means that, even beyond the technical inefficiencies of every L2 developer reinventing the wheel, siloed L2 implementations severely limit the usefulness of these technologies in practice.

Many open problems remain before shared L2 infrastructure can achieve feature parity with bespoke implementations today. Nonetheless, we believe

that it will become increasingly important as global adoption escalates and user needs diversify.

Acknowledgments

Our sincerest thanks to Aditya Asgaonkar, Ed Felten, Ethan Heilman, Syuhei Hiya, Kevin Ho, Liam Horne, Harry Kalodner, Georgios Konstantopoulos, Xuanji Li, Will Meister, Yuriko Nishijima, Georgios Piliouras, François-René Rideau, Dan Robinson, Nate Rush, vi, Jinglan Wang, Barry Whitehat, and Sebastien Zany for discussion and feedback while writing this work.

References

[1] *Perun: Virtual Payment Hubs over Cryptocurrencies.* Stefan Dziembowski and Lisa Eckey and Sebastian Faust and Daniel Malinowski. 2017. `https://eprint.iacr.org/2017/635`

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

[3] *Layer 2 state schemes.* Vitalik Buterin. 2019. `https://ethresear.ch/t/layer-2-state-schemes/5691`

[4] *On-chain scaling to potentially 500 tx/sec through mass tx validation.* Vitalik Buterin. 2018. `https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-thro`

[5] *A scalable verification solution for blockchains.* Jason Teutsch and Christian Reitwießner. 2017. `https://people.cs.uchicago.edu/ teutsch/papers/truebit.pdf`