Language-Agnostic, Upgradable Smart Contracting for Enterprise Applications

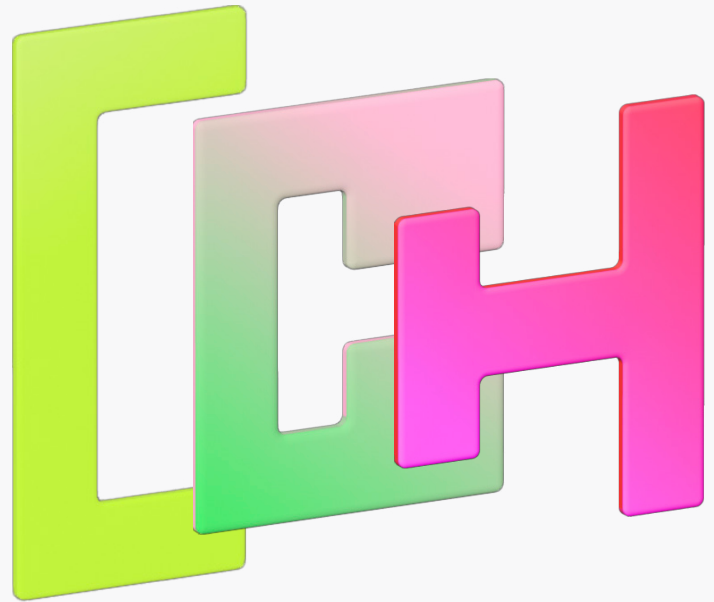
Despite increasing adoption of blockchain technology, decentralized applications remain isolated from traditional software development and established institutions. This trend is partly due to proprietary programming languages and the difficulty of incorporating blockchain solutions into existing infrastructure.

Casper's use of WebAssembly (Wasm) allows most common programming languages to work directly with any instance of a Casper blockchain, avoiding the hurdle of pre-existing software development teams requiring an onboarding time to learn new languages. So long as it can compile to Wasm, they may use their language of choice for developing Casper applications.

Introduction

The proliferation of blockchain technology over the past decade proved an existing need for a more robust, secure remote computing solution. While cloud computing and centralized servers fill one need, decentralized systems provide an alternative with several specific differences. The nature of a blockchain relies on disparate parties validating any business logic conducted on chain.

On a Casper network, this takes the form of individual nodes running in tandem to support a unified global state. Global state on a Casper blockchain exists as a Merkle trie. Casper's implementation of the Merkle trie key-value storage data structure allows any data to be shared individually in a verifiable way through a Merkle proof.



Casper uses an account-based, Proof-of-Stake (PoS) model that performs execution after consensus. In effect, changes to global state occur after a certain number of validator nodes reach consensus. This number varies based on the cumulative weight of CSPR staked by the nodes in question.

Users may interact with global state by sending a Deploy to a Casper network. A Deploy contains session code consisting of compiled Wasm and the sender's signature. If executed successfully, the Deploy will cause some degree of change to global state, be it installing contract code, interacting with a previously installed contract, or simply transferring motes between purses. A Casper blockchain does not natively recognize a CSPR, which exists as a convenience for users. A single CSPR contains one billion motes.

Smart contracts may be installed permanently to global state and accessed by any account with the proper permissions. These permissions take the form of an Unforgeable Reference or URef, with variable access levels. Further, the smart contract package structure allows direct upgrading while maintaining the same entry vector for third-party access. A single contract package hash will consistently point to the most recent version of a contract installed. All previous versions remain visible, preventing malicious actions by bad actors.

Casper features multi-signature approvals with weighted keys as an additional access control measure. Action thresholds restrict what actions an account can take based on the weight of their signature. These granular security measures, along with the use of Wasm as a native language, make Casper one of the most flexible blockchain platforms available.

Casper Design Features

1. Network Architecture

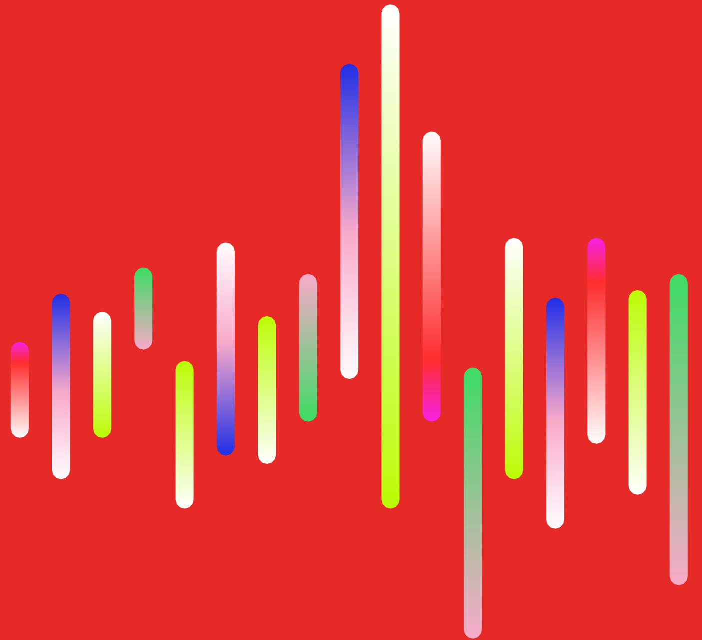
Global State

All accounts, contracts, and associated data on a Casper network are stored in global state. As explained above, global state exists as Casper's storage layer and can be represented as a Merkle trie key-value store.

Casper's trie features a radix of 256, with each branch node having up to 256 children. A path through the tree can be an array of bytes, with serialization linking a key with a path through the tree as its associated value.

A trie node can be one of the following types:

- A leaf, which includes a key and a value.
- A branch, which has up to 256 blake2b256 hashes, pointing to up to 256 other nodes in the trie.
- An extension node, which includes a byte array (called the affix) and a blake2b256 hash pointing to another node in the trie.



The extension node allows for path compression. Rather than traveling through a series of branch nodes, the extension node can feature an affix equal to the initial bytes in the series and a point to the first non-trivial branch node in the sequence.

When interacting with a Casper network, accounts are either viewing data stored in global state, attempting to install additional code in the form of a contract, or interacting with previously installed contracts. Any changes to global state occur through the finalizing of blocks to be added to the overall chain, creating a historical account of all alterations made since the genesis of the Casper network in question.

Unforgeable References

Unforgeable references are a key type representing the majority of data in a Casper blockchain – with the notable exception of Accounts.

casper

Referred to as URefs, unforgeable references carry permission information to prevent unauthorized use of the value stored under their key. The runtime tracks this permission information and will raise a forged URef error if malicious Wasm attempts to produce a URef without appropriate permissions.

Blocks

A block is the primary data structure through which network nodes communicate information about the state of a Casper network. Casper networks exist as a blockchain, a perpetual continuation of blocks added sequentially and representing immutable historical data.

Each block consists of a block header, a body, and a block hash derived from the blake2b256 hash of the block header.

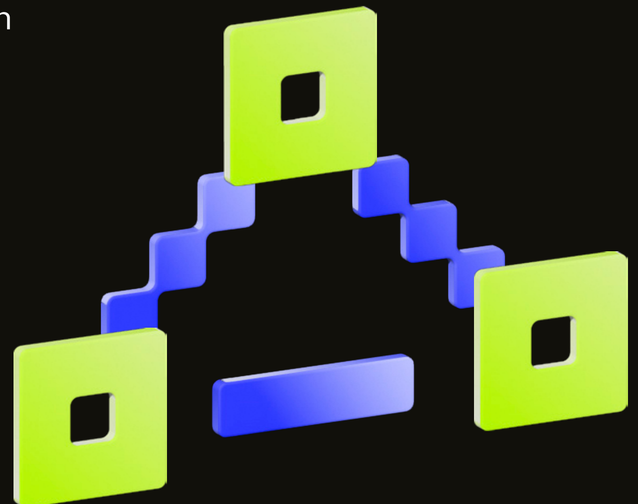
The body contains an ordered list of hashes for deploys and native transfers. These deploys can be broadly categorized as some unit of work that, when executed and committed, affect change to global state.

The block header contains a variety of information necessary for continued blockchain operations, as well as the hash of the body as described above.

2. Network Communications

Nodes

Nodes are the backbone of any Casper network. Both validating and read-only nodes provide robustness and decentralization to the overall network, ensuring proper operation and efficient throughput of client deploys.



Each node can be identified by its IP/Port pair where it is reachable. This address may also be referred to as an endpoint. When a node connects to another node with a known endpoint, it will open a TLS connection to the endpoint's address.

During connection setup, the client node will verify known information about the remote node and vice-versa, ensuring that all certificates are valid and each node is who it claims to be. However, only the node initiating the connection will send any messages.

After this initial connection, the node will enter Node Discovery and attempt to connect to any other node on the network, leading to a fully connected network.

casper

Gossiping

Gossiping is a process of distributing a value across the entire network without directly sending it to each node. Sending a deploy to a node and having that node accept the deploy causes it to be gossiped by that node to all other connected nodes.

Over time, information gossiped between nodes will reach a high enough saturation of overall nodes to be considered canon to the network and added to a block for finalization, execution, and addition to the overall chain.

Consensus

Consensus is a state where a certain critical number of nodes determine the `truth` of the blockchain moving forward. This continuous, trust-less process finalizes blocks and adds them to the chain in question, thereby perpetuating the blockchain.

Casper networks achieve consensus through a Proof-of-Stake or PoS method that determines trusted validators based on their investment or stake in the network's success.

Validator Auction

A node that the network recognizes as part of this process is known as a validator node. Validators are chosen through an auction process at the end of each era. The last block in an era is known as a switch block and includes additional data when compared to a regular block.

Validators chosen through this auction become validators for a future era, the exact timing of which depends on the internally set auction delay. Casper's Mainnet uses a delay of one, resulting in the chosen validators coming into effect after a full era has passed.

Nodes seeking to become validator nodes must bid an amount of CSPR that will be held by the system auction contract as their staked interest. Individual users may also delegate their CSPR to a validator to further increase the available stake of CSPR. These delegators will also receive a portion of the consensus rewards, consensus rewards relative to the tokens they delegate.

Among the validators, the network also chooses a single node to act as the lead validator for the round in question. This lead validator proposes new blocks to be added to the chain, which will then be gossiped to other nodes until they reach a critical mass majority. This occurs when the interconnected messaging can no longer switch to a different block, after which that block can be considered finalized.

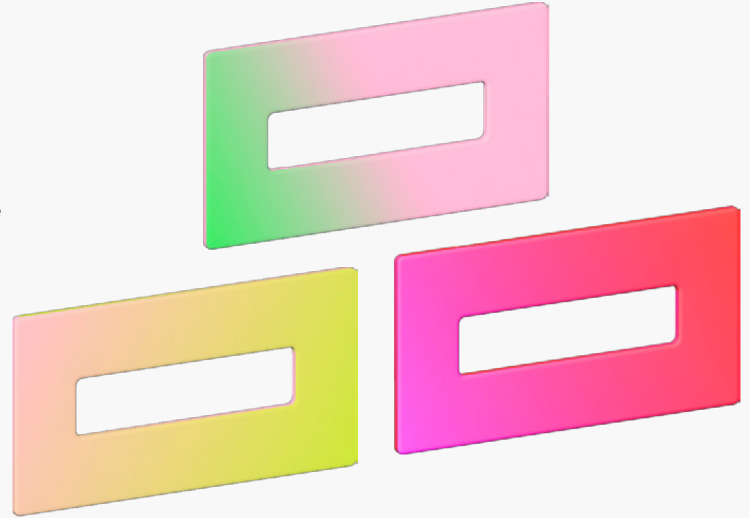
Any transactions included within a finalized block can also be considered as finalized.

3. Accounts

Account Creation

Casper networks use an account-based model, in which each account can be uniquely identified by an Account Hash derived from a Public Key. This identification method allows Casper to support a variety of key variants by standardizing key length.

Currently, Casper supports both Edwards-curve Digital Signature Algorithm, or Ed24419 keys, and Elliptic Curve Digital Signature Algorithm, or Secp256k1 keys.



An account uses these keys to sign deploys sent to the network, allowing other entities to identify their origin point.

Account creation on a Casper network involves creating a key pair of one Public Key and one Secret Key. Signing a deploy or interacting with the Casper network requires both keys. However, simply creating a key pair does not create a Casper account.

Account creation requires funding the main purse associated with the account's public key.

Multi-Signature Functionality

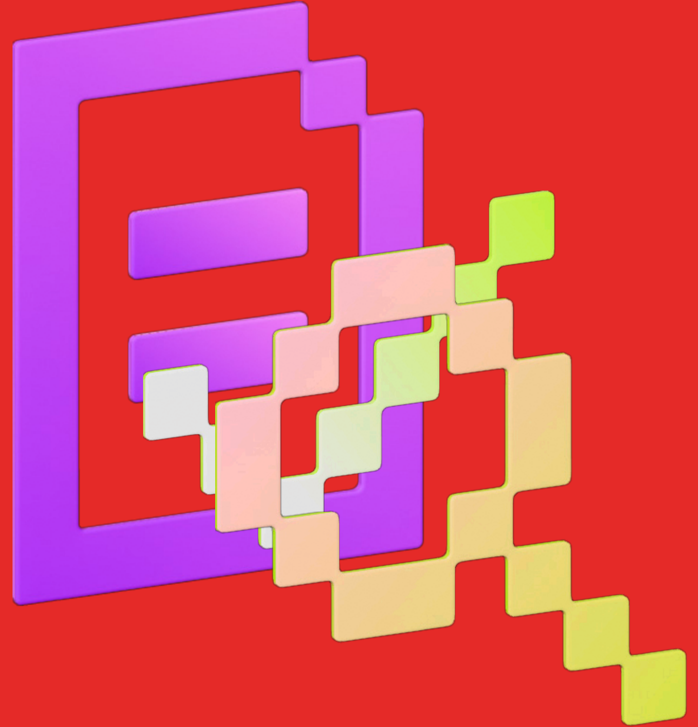
Casper networks natively feature multi-signature functionality that allows granular security access via associated keys.

Users can assign an action threshold to their account's ability to send deploys or manage keys. Any associated key with a weight higher than the threshold can perform these actions.

Alternatively, several associated keys with a combined weight higher than the threshold can sign the deploy or alter key management aspects. Different thresholds, weights, and number of associated keys can create a variety of security scenarios as required for a given operation.

4. Deploys

Deploy Structure



Deploys are a data structure containing Wasm and the signature of the sending account. They are the method by which users can request changes to global state. A deploy contains session code executed in the caller's context, which can, in turn, install or interact with contract code on the network itself.

A deploy consists of a body containing payment and session code and a header containing the public key of the account, which will serve as the context for the deploy, a timestamp of the deploy's creation, a time-to-live value that determines potential expiry, and a blake2b256 hash of the body as listed above. Further, it includes a deploy hash that is a blake2b hash of the header and a set of approval signatures that have signed the deploy.

A successful deploy execution results in permanent changes to global state and will incur a proportional cost in CSPR based on the complexity and size of the deploy in question.

Deploy Lifecycle

Deploy Received

Clients first send their deploy to one or more nodes using the node's JSON-RPC server. The node immediately checks that the deploy adheres to configuration settings outlined in the network's chainspec.

If the node accepts the deploy as valid, the system returns a deploy hash to the client that can be used to monitor the status of the deploy in question. If a deploy reaches or exceeds the time-to-live setting, it will expire.

Deploy Gossiped

After accepting a new deploy, the accepting node will begin to gossip the deploy to all other nodes. This process ultimately leads to a validator node placing the deploy into the block proposer buffer, where the lead validator chooses deploys to create a new proposed block.

Block Proposed

The lead validator for the round proposes a new block to be added to the chain, which includes the maximum number of deploys from the block proposer buffer that fit in a block.

Block Gossiped

As with the deploy gossiping, the lead validator begins gossiping the proposed block to all other nodes.

Consensus Reached

Consensus occurs when the current set of validator nodes conclude that the proposed block is valid. After reaching consensus, all included deploys are executed, and the block is added to the end of the chain.

Deploy Executed

Deploy execution occurs in three parts: payment code, session code, and finalization.

Payment code occurs first and determines the required payment amount for the requested computation included in the deploy. Only accounts with a minimum balance greater than the maximum potential payment code cost can execute deploys. If this is not the case, the system will revert any changes and deduct a penalty payment from the sending account's main purse.

If the purse holds sufficient motes, the system will execute the session code that includes the main logic to be computed.

Assuming both the payment code and session code are successful, the deploy is finalized and changes made to global state become immutable.

Execution Semantics

External FFI

Casper's use of Wasm enables greater flexibility when choosing a programming language for development within the ecosystem. Any programming language that can compile to Wasm will work with Casper networks; however, Wasm modules cannot natively create any effects outside of reading or writing from their own linear memory. These functions must be imported from the host environment.

In Casper's case, this is accomplished through an External Foreign Function Interface (FFI) containing low-level bindings for host-side functions, which in most cases should not be used directly. The Casper contract API provides higher-level bindings for writing smart contracts.

The Casper External FFI contains the following functions for use in Wasm:

casper_add: Adds the provided value to the current value under the provided key in global state.

casper_add_associated_key: Attempts to add the given public key as an associated key to the current account.

casper_add_contract_version: Adds a new contract version to a contract package.

casper_blake2b: Returns a 32-byte BLAKE2b hash digest from the given input bytes.

casper_call_contract: Calls a contract by its hash.

casper_call_versioned_contract: Calls a contract by its package hash. Optionally accepts a serialized contract version to call; otherwise, the most recent version.

casper_create_contract_package_at_hash: Creates a new contract at hash. This function returns the new contract package hash and a URef for modifying access.

casper_create_contract_user_group: Creates a new named contract user group under a contract package.

casper_create_purse: Uses the mint contract to create a new, empty purse.

casper_dictionary_get: The bytes in Wasm memory will be used together with the current context's seed to form a dictionary. The value at that dictionary is read from global state, serialized, and buffered in the runtime.

casper

casper_dictionary_put: The bytes in Wasm memory will be used together with the current context's seed to form a dictionary. This function writes the provided values under the dictionary in global state.

casper_disable_contract_version: Disables a version of a contract within a contract package.

casper_get_balance: Uses the mint contract's balance function to return the balance of the specified purse.

casper_get_blocktime: Gets the timestamp of the block in which this deploy is included.

casper_get_caller: Returns the public key of the account for this deploy.

casper_get_key: Returns the value written under the specified key.

casper_get_main_purse: Returns the main purse of the supplied account.

casper_get_named_arg: Copies the content of the current runtime buffer into the Wasm memory, beginning at the provided offset.

casper_get_named_arg_size: Queries the host side to check for a given named argument and returns its size in bytes.

casper_get_phase: Writes bytes representing the current phase of the deploy execution to the specified pointer.

casper_get_system_contract: Returns a system contract by name.

casper_has_key: Returns `true` if the key `name` exists in the current context's named keys.

casper_is_valid_uref: Checks if all the keys contained in a provided value are valid in the current context. (For example, the value does not contain any forged URefs.)

casper_load_call_stack: Load the URef known by the given name into the Wasm memory.

casper_load_named_keys: Returns the named keys from a contract's context.

casper_new_dictionary: Creates a new URef that points to a dictionary.

casper_new_uref: Causes the runtime to generate a new URef, with the provided value stored in global state.

casper_provision_contract_user_group_uref: Requests that the host provision additional URefs to a specific group identified by its label.

casper_put_key: Writes the given value under the specified key.

casper_read_host_buffer: Copies the contents of the current runtime buffer into the Wasm memory, beginning at the provided offset.

casper_read_value: Deserializes the bytes in the span of Wasm memory of the key provided.

casper_record_era_info: Records era info. It can only be called from within the auction contract.

casper_record_transfer: Records a transfer. It can only be called from within the mint contract.

casper_remove_associated_key: Attempts to remove the given public key from the associated keys of the current account.

casper_remove_contract_user_group: Removes a group from the given contract package.

casper_remove_contract_user_group_urefs: Removes a user group's URefs.

casper_remove_keys: Removes the specified keys from the associated keys set.

casper_ret: Terminates the currently running module after copying bytes to a buffer, which is returned to the calling module.

casper_revert: Terminates the currently running module while reverting all effects of the execution up to this point.

casper_set_action_threshold: Changes the threshold to perform the specified action.

casper_transfer_from_purse_to_account: Uses the mint contract's transfer function to transfer tokens from the specified source purse to the target account.

casper_transfer_from_purse_to_purse: Uses the mint contract's transfer function to transfer tokens from the specified source purse to the target purse.

casper_transfer_to_account: Uses the mint contract's transfer function to transfer tokens from the current account's main purse to the main purse of the target account.

casper_update_associated_key: Attempts to update the given public key as an associated key to the current account.

casper_write: Writes the provided value under the provided key.

More in depth information on these functions can be found in Casper's technical documentation.

casper