IOTA Smart Contracts

Evaldas Drąsutis

IOTA Foundation (evaldas@iota.org)

November 10, 2021

Abstract

The document introduces IOTA Smart Contracts, a distributed ledger technology (DLT) and a multi-blockchain smart contract framework with ability to transact cross-chain in a trustless and scalable manner on the IOTA UTXO ledger on layer 1. The main goal is to present the reasoning behind the concept and the architectural elements. The style of the whitepaper is informal.

Disclaimer. Design decisions presented in the whitepaper may change in the future. Some topics are preliminary plans and are under active research. There's no guarantee that the actual development path will follow the guidelines of this document in detail.

Many thanks to Levente Pap, David Hagman, David Phillips, Jorge Silva and Gunnar Stenzel for feedback, numerous suggestions and edits of the text.

Contents

1 Summary **3**

2 Goals and use cases **3**

3 Rationale **4**

3.1 Blockchain smart contracts 5

3.2 UTXO smart contracts 6

3.3 IOTA's approach to smart contracts 7

3.3.1 IOTA DLT 7

3.3.2 Multiple layer 2 ledgers anchored on layer 1 8

3.3.3 Validation of ISC chains 9

3.3.4 Validity of the state transition 11

3.3.5 Trustless cross-chain transactions 12

3.3.6 Extension of the UTXO Ledger 12

4 Validation incentives **12**
4.1 Fees . . . 12
4.2 Feeless . . . 13
4.3 Incentives to validate L1 . . . 13

5 Security considerations **14**
5.1 Introduction . . . 14
5.2 Consortium . . . 15
5.3 Corporate or DAO setup . . . 16
5.4 Shared security: multiple chains with shared validators . . . 16
5.5 Shared security: a permissionless setup . . . 17

6 The IOTA Smart Contract chain **18**
6.1 Smart contract chains . . . 18
6.2 Chain account and Chain ID. . . . 19
6.3 State. Anchoring the state . . . 20
6.4 Committing to the state . . . 22
6.5 The state transitions . . . 24
6.6 State synchronization and validation . . . 25
6.7 On-chain ledger . . . 26
6.7.1 Smart contracts . . . 26
6.7.2 On-chain accounts . . . 26

7 Nodes **28**
7.1 Validator nodes. Access nodes . . . 28
7.2 Consensus . . . 29

8 The virtual machine **30**
8.1 VM abstraction . . . 31
8.2 Core contracts . . . 31
8.3 Gas and fees . . . 32
8.3.1 Gas considerations . . . 32
8.3.2 Fees . . . 33

9 Governance of the chain **33**

10 Invoking smart contracts. Composability **34**
10.1 Calling smart contracts . . . 34
10.2 Requesting smart contracts . . . 35
10.2.1 On-ledger requests . . . 36
10.2.2 Asynchronous composability . . . 36
10.2.3 Off-ledger requests . . . 37
10.3 Tokenization. Bank and cash metaphor . . . 38

11 Annex: Assumptions about L1 UTXO ledger **39**
11.1 Introduction . . . 39
11.2 Scalability of the UTXO ledger . . . 41
11.3 Ledger, ledger state and pruning . . . 41
11.4 UTXO ledger accounts . . . 41
11.5 UTXO extensions . . . 42

1 Summary

In the whitepaper we introduce the rationale behind IOTA Smart Contracts (ISC) and present its main architectural decisions. ISC is a part of the IOTA[1] *distributed ledger technology* (DLT).

ISC is a **framework** that extends the base protocol of layer 1 (L1) of the IOTA DLT by introducing multiple programmable ledgers on top as layer 2 (L2). The result is a multi-chain environment where all chains are anchored on the IOTA Ledger at L1. Each chain is a separate blockchain with smart contracts on it, functionally equivalent to Ethereum smart contracts and fully composable between each other. The ISC environment also enables the trustless transacting and composability of smart contracts between different chains over L1 with high throughput and high scalability. Theoretically, the ISC as distributed and sharded multi-chain DLT which encompases both L1 and L2 can scale up to hundreds of thousands of smart contract transactions per second.

The ISC concept also includes a significant extension of the IOTA Unspent Transaction Output (UTXO) ledger with configurable UTXO types and **advanced tokenization** features. The combined L1 and L2 concepts with advanced tokenization features make ISC a unique combination of UTXO ledger, multi-chain scalability, and quasi-Turing complete programmability.

This whitepaper is a condensed presentation of the efforts made on ISC in research and several working prototypes and releases during 2019-2021. It represents the state-of-the-art of the ISC concept of the early beta release of ISC in October 2021 for the experimental IOTA 2.0 DevNet. The whitepaper describes the concepts, goals, rationale and architecture of ISC in a generic manner, independent of a specific version of the base IOTA protocol.

IOTA Smart Contracts are implemented as **Wasp node software**[2]. This document may include particularities or even implementation details of certain features. This document also includes preliminary statements and features that are planned to be implemented in the future and/or are in active research. There is no guarantee that those features will be implemented the way they are described in this whitepaper.

2 Goals and use cases

Scalability is the main challenge in the DLT realm, as well as scalability-related problems such as congestion and the ever-growing cost per transaction. In its original form, blockchain is not a scalable setup. There is a growing conviction in the market that **the future of the DLT and crypto industry is multi-chain and multi-ledger**. The multi-chain vision of DLT is often associated to *sharding* of L1 by introducing multiple parallel L2 ledgers.

Multi-ledger, however, also means that each distributed ledger is secured by its own set of validators (or 'miners'). This fact makes it difficult to transfer value between two ledgers/blockchains without a trusted party in between, such as a *bridge* or a *relay*. The

[1] IOTA website: https://www.iota.org
[2] Wasp node GitHub repository: https://github.com/iotaledger/wasp

cross-chain gap also breaks the comfortable composability of smart contacts on the same blockchain, and this is the main limitation of multi-chains systems.

To address these challenges, the IOTA Foundation's ISC aims to provide a universal framework and platform for **multi-chain DLT** hosting **parallel yet composable smart contracts**, enabled by a native ability of **trustless cross-chain transactions** which use the scalable IOTA ledger as a backbone. With ISC, the IOTA Foundation aims to build highly scalable distributed ledgers with real-world industry requirements and throughput of hundreds of thousands of writes and value transfers per second.

ISC is designed with the following broad classes of use cases in mind:

- **Consortium deployments** of multi-chain smart contract networks running on top of the public IOTA network.
- Open **permissionless multi-chain environments** and ecosystems on top of the public IOTA network. Scalable public decentralized apps (dApps), needed for example for Decentralized Finance (DeFi).
- **Corporate deployments**, for both public and private institutions, private, semi-closed and open deployments, such as distributed ledgers at the pan-European and global scale for public, energy, logistics, food and other industry verticals.

ISC also aims to provide a comprehensive framework for the **tokenization** of any on-ledger assets in the multi-chain/multi-ledger environment. Any smart contract asset on any chain, be it ERC-20 token, sensor reading of the solar panel, or art NFT, can be tokenized and transacted between multiple smart contracts on multiple chains, without any additional trust in between.

ISC offers a multi-ledger paradigm. Each ledger, whether lightweight or heavy, is a module composable with others to build systems of arbitrary scale. In this paradigm, we see the L1 ledger together with multiple ledgers on L2 as one protocol with huge scalability and potential for different use-case-specific configurations.

3 Rationale

ISC aims to extend the base IOTA distributed ledger protocol, a scalable, parallelizable, fee-less and miner-less DLT, with **distributed trustless computations on the ledger**. Here we provide the rationale on how we achieve this goal and why we chose this approach.

A smart contract is essentially a state machine that, upon external input such as a trigger or a call, takes the **ledger state** and produces an **update** to it. The state machine, its program, is part of the ledger itself. The smart contract becomes an extension of the ledger state transition function. Instead of having a hard-coded logic of the transaction as the only option of state transition, smart contracts enable an extension of the static transaction concept with the pluggable logic of the ledger state transition function. The computational model of the pluggable extensions may be in the range starting from limited configurable logic and scripting, like Bitcoin scripts[3], up to the rich quasi-Turing complete

[3]Bitcoin script: https://en.bitcoin.it/wiki/Script

programming environments offered by Ethereum Virtual Machine[4] (EVM) and Solidity[5]. With ISC, we are aiming at the latter.

Let's look at existing paradigms of the programmability of ledgers.

3.1 Blockchain smart contracts

Smart contracts are programs run by a distributed processor, i.e. many processors reach consensus on the result of computations. The essential requirement for smart contracts is the objective (ledger) state, i.e. a state equally perceived by all smart contracts updating it. *Objectivity* here means that all nodes running the distributed processor have full agreement on the current ledger state, the main input for computations.

Blockchain, a totally ordered chain of ledger updates, achieves the **global objective state** by serializing state updates to the ledger through a strictly sequential chain structure and distributed consensus mechanism. The consensus mechanism allows several competing versions of the global state at the same time, before eventually eliminating all except one.

There are two well known models of the ledger in the DLT space: the **UTXO ledger model** and the **account-based ledger model**.

The UTXO ledger consists of UTXOs (*unspent transation outputs*, see Annex: Assumptions about L1 UTXO ledger) which are grouped into accounts by the target address. Therefore, the asset balance of the UTXO account is a sum of balances over multiple UTXOs. The essential property of the UTXO ledger is that it allows **parallel writes**, i.e. parallel updates of the state of the account. In this case, for a consumer of the account it makes the state of the UTXO account non-deterministic.

In the account-based model, accounts and associated balances are global atomic entities, which can only be updated sequentially. This means that the account balance in an account-based ledger is deterministic (or objective).

While blockchains use both UTXO (such as Bitcoin[6] and Cardano[7]) and account-based (such as Ethereum[8]) ledger models to represent the global objective state, the account-based model with global accounts is natural for blockchains, while putting UTXOs into blocks cancels the parallel nature of the UTXO ledger.

The existence of a global and objective state immediately makes the **blockchain a natural environment for smart contracts**. A blockchain ledger makes the perception of the ledger state exactly the same (objective) for all smart contracts, updating it in the same block.

The blockchain smart contracts are usually quasi-Turing complete, i.e. they can implement any algorithms by assuming unlimited time and memory resources. For this, they can use universal computation models and languages such as EVM/Solidity, WebAssembly

[4]Ethereum Virtual Machine: https://ethereum.org/en/developers/docs/evm/
[5]Solidity: https://docs.soliditylang.org/en/v0.8.9/
[6]Bitcoin: A Peer-to-Peer Electronic Cash System: https://bitcoin.org/bitcoin.pdf
[7]Cardano UTXO https://docs.cardano.org/plutus/eutxo-explainer
[8]Ethereum WP: https://ethereum.org/en/whitepaper/

and so on. The *quasi* signifies that the run space of those programs is actually limited at runtime on purpose, usually by introducing the concept of **gas**, in order to prevent abuse of computer resources.

The synchronous perception of the same state by blockchain smart contracts makes them highly **composable**, a feature actively used in blockchain ecosystems like DeFi.

At the same time, the sequential and synchronous structure of blockchains also introduces the fundamental **bottleneck** of blockchain-based DLTs into smart contracts and is ultimately responsible for their **limited scalability**. In simple terms: **you can't run parallel smart contracts and parallel state updates on the global state of the blockchain**. Efforts to run them faster and faster do not take away the fundamental bottleneck, while the sharding usually splits the global ledger state into sub-states and, strictly speaking, the ledger stops being a blockchain.

As it is explained below (see IOTA DLT), the DAG structure of the IOTA ledger and IOTA consensus does not allow to have global objective state of the ledger's account for the nodes. This fact is fundamentally related to the parallelism and scalability of the IOTA L1 ledger. Absence of the global objective state on the IOTA ledger prevents IOTA from implementing the model of blockchain smart contract on its L1 directly.

3.2 UTXO smart contracts

UTXO smart contracts (also known as *UTXO scripts*) is a computational model which limits the run space of the program to one transaction. Due to the parallel nature of the UTXO ledger, UTXO smart contracts run parallel on independent branches of the state.

There are well-known examples of UTXO ledger programmability on L1 with UTXO smart contracts; for example, Bitcoin scripts or Cardano's EUTXO model[9]. It is straightforward to implement these on any UTXO ledger, including the IOTA UTXO ledger.

One might ask: why not use this approach for adding programmability to the IOTA ledger?

The short answer is that we follow this model for certain programmability of L1, in order to support ISC and advanced tokenization. However, as a universal approach to smart contracts, it is too limited for real-world industry use cases. We will elaborate on the reasons for this.

The **throughput** of UTXO smart contracts is limited by the throughput of the L1 itself. Besides, UTXO scripting introduces significant overhead on top of the throughput of the base layer, so it does not meet the requirements of the real-world industry.

The **computational model** of UTXO smart contracts is too limited compared to the expressiveness and richness of quasi-Turing complete smart contracts, which is the mainstream option in the blockchain ecosystems. The best example is Ethereum with EVM and Solidity.

[9]The extended UTXO model: https://iohk.io/en/research/library/papers/the-extended-utxo-model/

A UTXO smart contract (also known as UTXO script) does not have access to the global state and cannot make assumptions about the global order on the ledger (otherwise it will not be scalable or it will lead to a need for centralization; for example, delivery of scripts to each L1 validator and similar). Therefore, the state consumed and produced by a UTXO smart contract is fundamentally limited by the boundaries of the transaction. This prevents UTXO scripts from using the usual machinery of full-scale programming, like the one offered by EVM and Solidity. Instead, the programming of realistic dApps on UTXO scripts requires unorthodox approaches and tools (including languages) to represent the computations as a collection of functionally limited finite state machines and data structures. This approach results in significant performance and user experience (UX) penalties.

While the overall UTXO programmability approach offers notable strong points, including verifiability of UTXO scripts, in our opinion this does not justify the practical disadvantages in many (or most) real-world use cases. Many industry use cases, including DeFi, need unbounded data structures, complex algorithms, good UX, scalability and high throughput.

Fee-less nature of the IOTA ledger. The absence of fees in the IOTA ledger prevents us from introducing full-scale programmability through UTXO scripting because it does not enable the easy introduction of gas and gas budgets concepts for computations on L1. This is a necessary feature to prevent the abuse of processor and storage resources by smart contract programs.

3.3 IOTA's approach to smart contracts

Removing the scalability bottleneck is one of the main goals of the IOTA DLT. With ISC, we aim to remove bottlenecks in the smart contracts system via horizontal scaling.

In short: we aim to enable **parallel smart contracts without losing composability**. To achieve this, we combine the two approaches, the blockchain smart contracts on L2 with advanced UTXO types on L1, in one framework.

3.3.1 IOTA DLT

IOTA overcomes known scalability bottlenecks in its distributed ledger by using partially ordered data structures based on *Directed Acyclic Graph (DAG)*, instead of the strictly sequential and totally ordered structure of the blockchain.

The DAG structure appears in IOTA in both the ledger and the consensus layers. Two essential components of IOTA protocol together differentiate it from blockchain platforms:

IOTA Ledger, also known as **UTXO ledger** (or sometimes called *UTXO DAG*, even if it is a more complex data structure than a DAG). For a UTXO transaction, there is no need to access the global state of the account, so **parallel updates (or 'writes') to the ledger** become possible. It is a necessary condition for scalability and high throughput.

IOTA Tangle, a permissionless, feeless and miner-less consensus protocol based on DAG[10]. The Tangle consensus protocol is based on validation of the ledger by the users of

[10]IOTA research papers: https://www.iota.org/foundation/research-papers

the ledger themselves, not by miners. The user adds transaction to the ledger by validating two already *attached* transactions and their past cones, and then *attaches* its transaction to those two, thus extending the Tangle by voting on selected transactions. The equilibrium in the consensus comes from the assumption that optimal strategy for the (honest) user is to attach its transaction so that to maximize its chances to be picked by others, and that is possible only if you attach to the valid ledger in the history.

The IOTA Tangle enables global consensus on the UTXO ledger state without the need to impose a linear order among unrelated ledger state updates (transactions). That allows consensus on the state of the account without compromising the scalable nature of the UTXO ledger.

The two components above can't live without each other. The combination of the two gives the IOTA DLT a unique feature regarding scalability: in theory, an unlimited number of parallel writes to the ledger.

Note that blockchain-based consensus protocols, if applied to the UTXO ledger model, cancel out scalability advantages offered by the latter, by locking the inherently parallel structure of the UTXO Ledger into the blocks and strict sequence of the chain.

IOTA approaches the scalability problem by enabling parallel updates of the global distributed ledger. This makes the approach fundamentally different from many blockchain platforms which use classical Nakamoto or BFT consensus blockchain systems on L1, no matter if their ledger model is UTXO based, or based on a system of global accounts.

The scalability advantages lead to inevitable trade-offs, namely the impossibility for a deterministic program to assume any global state. An account on the UTXO ledger is a collection of UTXOs with the same target address. It can be updated by asynchronous entities in parallel, therefore it may be perceived differently by different parties. This makes the state of the UTXO account (and the whole ledger state) subjective, i.e. different parties (for example, smart contracts) do not always have an equal view of it.

The absence of an objective global state of an account **prevents IOTA from implementing quasi-Turing-complete programmability of the L1 in the form of classical blockchain smart contacts**.

3.3.2 Multiple layer 2 ledgers anchored on layer 1

IOTA choose to build multiple totally ordered ledgers on top of the IOTA Ledger, essentially layer 2 (L2) blockchains, in combination with extended functionality of the UTXO ledger on L1 to support the L2 multi-chain environment through state anchoring and advanced tokenization features. This achieves the following:

- Quasi-Turing complete smart contracts with rich expressiveness and composability on each L2 chain.
- Scalability of the system due to massive parallelization through **multiple parallel** ledgers (also known as 'shards') and **parallel smart contracts**.

Each such chain will host many smart contracts, all of them sharing the same objective state of the chain. This makes it possible to run quasi-Turing complete smart contracts

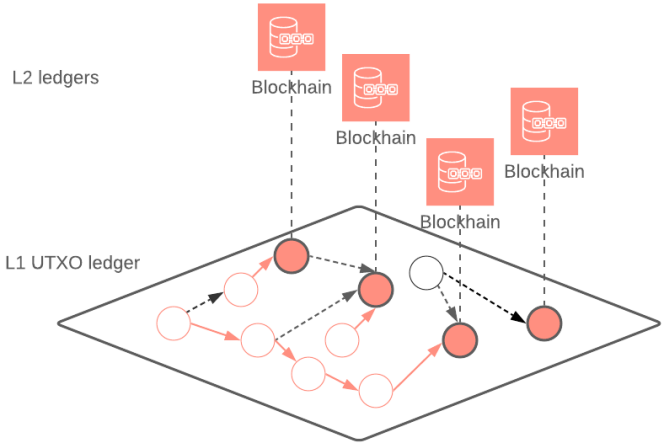

Figure 1: Multiple L2 chains anchored on L1

on it (including the original EVM contracts) and, at the same time, control assets on the UTXO ledger on L1. Each chain will also be able to process high throughput of transactions sent directly to the chain.

The scalability of the multi-chain system is enormous. For example, what would be the configuration with 100,000 TPS of throughput? Let's conservatively assume each such chain is capable of processing 200 transactions per second (e.g. sensor readings) **in parallel**. So, 500 chains will be able to settle a total of 100,000 readings per second into their ledgers.

How many resources will those 500 chains consume on L1? Each chain produces and confirms one state anchor transaction (or a 'block') per 10 seconds. All of those 500 chains will anchor their states on the L1 ledger once per 10 seconds, so they will consume only about 50 TPS on L1.

3.3.3 Validation of ISC chains

A **validator** is a real-world entity (a person or organization) that is **incentivized** to secure the ISC chain by validating it via participation in a **distributed consensus** with other validators. A validator is identified in the network with its public key; however, ISC is not explicit about how the validator is incentivized to validate the chain. We cover some possible options in Validation incentives.

The validator owns one or more **validator nodes** (currently implemented as Wasp nodes). Each ISC chain is run by its own set of validator nodes. Each validator node validates the chain **on behalf of the validator it belongs to**. So, whenever we talk about the behavior of a validator node, correct or faulty, the responsibility and motivation of such a behavior are linked to the validator, a real world entity.

To run an ISC chain, validators form a finite group called a **committee of validators** and delegate their validator nodes to it. The committee of validators runs a chain by **collectively** taking control of the L1 account (known as the 'chain account') as one of

the inputs and produces updates to the account.

- The committee of validators runs a *Byzantine Fault Tolerant* (BFT) consensus protocol on state updates/blocks. This makes an ISC chain and smart contracts on it a fault-tolerant and distributed system, in which $\lfloor 2N/3 \rfloor + 1$ of non-faulty validators is enough to produce valid state updates (here N is the number of validators if the committee).
- The committee of validators can be rotated depending on the governance model. By **rotating the committee of validators**, validators can be deleted, added, or replaced.
- We enable the validators of the chain to control the account on L1 through **threshold signatures** as a proof of consensus among validators (see also alternatives approaches in Validity of the state transition). Each validator node in the committee owns its private key share which is used to produce a partial signature of the transaction. To produce a valid threshold signature for the transaction, a quorum of $\lfloor 2N/3 \rfloor + 1$ validator nodes is enough to cooperate. The cooperation of validator nodes and signature aggregation is a part of the consensus process.
- To become a committee of validators, nodes run a **distributed key generation** (DKG) process. After the DKG procedure, each validator securely generates and keeps in its possession a private key share known only by itself.

We may see each ISC chain as a distributed wallet run by a distributed set of validators: the "wallet" takes assets incoming to the account and updates the account according to a program.

As noted above, there is no direct way on the IOTA ledger (L1) to see the objective state of the account for such a "wallet" on different nodes. This makes the running of consensus on ISC different from classical blockchains (e.g. Bitcoin):

- In classical blockchain consensus, the validators (or 'miners') each compute their own version of the future state based on the assumed current state and a set of transaction proposals chosen by it (this also creates the phenomenon of *miner extractable value*, MEV). The current state is a valid state for other miners, so the deterministically-computed future state is valid for them too. Validators then come to a consensus on which version of the **future state** to use in the global ledger, for example by running the Proof of Work (PoW) competition.
- In ISC, the current state of the account is non-deterministic, so the validators must first reach a consensus on the **present state**. The result is consensus among validators on the current state of the account and on other non-deterministic inputs (such as the batch of requests to be processed). After consensus on inputs is reached, the rest is just a deterministically verifiable computation run by VM; therefore, each validator node can compute it itself.

So, to provide a global objective state for smart contracts, we extend the accounts on the IOTA Ledger by building (block)chains on top. The committee of validators of the ISC chain runs a consensus on the state of the account. The approach of extension of the UTXO account with the chain on top of it can also be seen as **L2 sharding** of the UTXO

ledger.

This way we can have as many **parallel ledgers** on L2 as the number of possible accounts on L1, i.e. theoretically unlimited. We call them *chains*, also known as *smart contract chains* or *ISC chains*.

Each chain controls **on-chain assets** deposited to the chain's account and **commits** its state to the same account on L1. We call this process **state anchoring**.

3.3.4 Validity of the state transition

By providing a valid threshold signature of the state anchor transaction, the chain provides **proof of consensus** (i.e. the proof that chain validators reached consensus on the state transition), because only a supermajority of $\lfloor 2N/3 \rfloor + 1$ of the validators can produce a valid threshold signature. Without a proof of consensus (a valid signature), the L1 ledger will not confirm the transition of the state by consuming state anchor output and producing the next state anchor transaction.

This fact places trust in the validity of the state transition entirely on the quorum of chain validators. Once the state anchor is confirmed on L1, the state transition is finalized and becomes immutable and irreversible. However, the L1 ledger cannot validate the state transition. The colluding majority of $\lfloor 2N/3 \rfloor + 1$ validators may produce an invalid state transition with a valid signature, for example, to allocate funds not according to deterministic prescriptions of smart contract programs. That would be an example of misbehavior, similar to a 51% attack on PoW blockchains.

Methods of preventing this kind of majority attacks are in the realm of game-theoretical security, such as the one used in Ethereum optimistic rollups. See Security considerations for more details.

Threshold signatures are a well-understood and simple approach to proof of consensus with great performance characteristics. It is our main approach for anchoring the state on ISC chains.

The alternative approach with powerful properties would be to enable control of the chain's account (and the state transition) by **validity proofs**, instead of proofs of consensus with threshold signatures. This is a direction based on zero-knowledge proofs. The validators collectively compute the next state of the chain and provide compressed cryptographical proof that the state transition was computed by deterministic algorithms of the virtual machine. The zero-knowledge proof is checked on L1. Its validity is a condition for the state transition to occur on L1.

The main challenge of such approach is the complexity of the virtual machine that computes the zero-knowledge proof and the fact that this process is computationally very heavy.

The valdity proof approach is being researched as a possible future direction for state anchoring in ISC.

3.3.5 Trustless cross-chain transactions

Through anchoring, the smart contracts on the ISC chain are able to own and manipulate L1 assets on the IOTA ledger according to their algorithms. The same mechanism enables smart contracts to exchange native IOTA assets with other smart contracts on other chains (i.e. cross-chain) without any additional trust or bridging.

The state of the smart contract chain is decoupled from the state of other smart contract chains. That makes smart contacts parallel (i.e. with asynchronous state) on different chains, yet enables cross-chain **interoperability** and **composability** through trustless transacting by utilizing IOTA's **trustless and feeless L1 asset transfers** between chains.

Inside one smart contract chain, we provide an environment at least functionally equivalent to a classical Ethereum smart contract platform, including full synchronous composability, based on synchronous access to the global state of all smart contracts on the chain. At the same time, multiple chains transaction over the same distributed ledger improves the global properties of the platform, scalability in particular.

3.3.6 Extension of the UTXO Ledger

To achieve the functionality described above, we significantly extend the UTXO ledger on L1. With this extension, we aim to achieve:

- Support of L1 for anchoring the L2 chains;
- Enabling of on-ledger transactions between smart contracts and chains; item Introduction of advanced tokenization features on L1 and L2 for cross-chain composability of smart contracts.

See more details in Annex: Assumptions about L1 UTXO ledger.

4 Validation incentives

As noted above, validators delegate their validator nodes to committees of validators in order to validate ISC chains on their behalf. Why would a validator spend its resources to run validator nodes?

In general, ISC is not specific about the exact mechanisms of incentives for validators, the way the incentive mechanism for miners is built into the Bitcoin protocol. ISC targets both permissionless and corporate environments, both fee-based and fee-less; therefore, the intention is to be open to all use cases. Here we discuss several options.

4.1 Fees

One option is the incentivization of validation based on **fee revenues**. Validators are paid **fees** via their validator nodes for resources provided. Usually the fees are collected from smart contract users (also known as *requesters*) and distrubuted to validators. Another possible model would be the owner (also known as *governor*) of the chain to remunerate validators directly for their provided resources. In any case, the motivation of the validator to provide resources to validate the chain is tied to the future expected revenue streams

from fees. Financially, it can be modeled as a *net present value* of the future expected income. The loss of such future income (for example due to the malfunction of the corresponding node) is an economic disincentivization (or punishment) for such behavior. The *fee revenue* is the main incentivization method for validation in a permissionless validation environment.

The fee collection mechanism in ISC is in-built on the virtual machine level. It is responsible for charging fixed and/or gas-based fees from requesters, i.e. the users of the chain's services. The charged fees are distributed among validators of the chain in a fair fashion. Due to the nature of the BFT consensus used, "fair" means that collected fees are distributed equally (on average over time) to the validators. However, certain contribution characteristics, such as availability rate and similar factors, may also be taken into account. The fee policy is defined by the Governance of the chain.

The collected fees are split into two optional parts: **validator fees** go to validators of the chain and **owner fees** go to the governor of the chain, for example a *Decentralized Autonomous Organization*, DAO.

4.2 Feeless

In the corporate or private setup, the validation of chains is incentivized by use-case-specific motives. The consortium members are the validators of the chain. So, motivations to run a validator node are identical to motivations to participate in the consortium. This kind of setup does not require any fee-based economical incentives, so the chain can be completely **feeless**. However, even in the closed setup, fees may be used in a way defined by the consortium policies, for example in order to prevent spam attacks and similar motives.

4.3 Incentives to validate L1

The user adding a transaction to the IOTA L1 ledger is also validating the L1 ledger: the submission of the transaction to the Tangle is also an act of validation of the past cone of the ledger, that is how IOTA Tangle consensus protocol works.

In the case of ISC chains, the *users* of the IOTA L1 ledger are validator nodes of the chain on L2. The validator nodes collectively produce an anchor transaction and post it to the L1. This way, each validator of an ISC chain (through the L2 validator nodes it runs) naturally has interest for transactions produced by its validator nodes to be confirmed on the L1. So the validator becomes a user of the IOTA L1 ledger and therefore has to validate it.

However, the ISC validator node on L2 cannot validate the IOTA L1 ledger itself, so it must trust the IOTA L1 node to which it is connected to attach its transactions to the Tangle. The only way to post the transaction to L1 **without needing to trust** a third party running the IOTA node is to **run the IOTA node by the validator itself**. It means each L2 validator has a clear motivation to run its own IOTA node as a trusted and secure access to the L1 ledger, in order to do a proper attachment to the Tangle and prevent eclipsing, downtime etc.

In other words, each L2 validator has a clear motivation to run its own IOTA node, thus becoming a validator for the IOTA L1 ledger.

5 Security considerations

ISC is a framework (or a tool) for building distributed ledgers; therefore, it is open to all kinds of configurations. In particular, it is not explicit about what model of game-theoretical or tokenomical security to use. The above overview of incentivization is the basis for different use-case-specific approaches to the question of security in the chain.

5.1 Introduction

The security of a distributed ledger depends on its validators. ISC chains are validated by finite (yet rotating) committees of validators. ISC is not specific about how exactly those committees are formed.

The committee of validators is a different validation setup from Proof of Work blockchains, where anybody can validate the ledger (in other words, produce the block) if they have enough hashpower. Permissionless validation offers high liveness and security but no real finality and limited scalability. It motivates the market to search for faster and cheaper alternatives.

In ISC, we uses committee-based BFT consensus, a well-understood option with excellent performance characteristics. The trade-off here is that nodes can't decide by themselves to be a validator for a chain. (It should be noted, however, that this does not prevent permissionless participation in the validation of chains. See Shared security: a permissionless setup).

We see ISC as a tool and ISC chains as building blocks for all kinds of systems, dApps and ecosystems. We see the right approach to security of validation as **use-case-specific**. It can be approached from different angles depending on what we want to achieve in each specific case.

The assumption of a ledger being *distributed* (fault-tolerant), *decentralized* and *trustless* usually means that we should not and do not trust any one particular validator to be honest. However, to a certain extent, we trust all validators as a whole. We give ourselves a reason to trust the distributed ledger by looking into its validators and their distributed consensus process on the ledger state. The trust may be seen as a perceived security in the distributed ledger, i.e. how safe we think our assets on that distributed ledger are.

The security itself can be modeled in different ways and from different angles. To simplify the matter, we assume the security of the DLT is defined by the chance that a certain majority of validators (depending on specific consensus mechanisms) will **collude** to coordinate their activity in order to produce an invalid state transition. An invalid state transition is a result of computations different from the one prescribed by the deterministic state transition function of the ledger. That is what "validators stealing my funds" essentially means.

In systems with permissionless validation running Nakamoto PoW consensus, like Bit-

coin or Ethereum, any entity can become a validator at its own will. The chances of collusion (51% hashpower) depend on dynamic situations (i.e. how deep is the concentration of hashpower) and game-theoretical factors, such as what is there to gain from the collusion among dominating validators (the mining pools). Systems with permissionless validation usually rely on a motivational feedback loop to attract more validators to make the system more secure (more expensive to attack) and, as a consequence, incur a higher total cost of running the system and scalability bottlenecks. Systems with permissionless validation usually are very trusted, even when it is enough to collude four or so top validators to take control over the state transition. The source of trust is the fact that any misbehavior is public, provable and traceable back to the colluding parties. The implicit game-theoretical reasoning behind this high level of trust is that, under such conditions, collusion will bring a net loss for the colluding parties.

In a so-called permissioned setup, a node cannot just decide by itself to become a validator for the ledger: it must fulfill certain conditions to be accepted as a validator. Many systems based on BFT consensus are permissioned in this sense. Many real-world systems are permissioned too, for example, while the representative democracy is permissionless (anybody can be elected), the parlament is permissioned (one must be elected by others).

In ISC chains, the committee of validators is a distributed and fault-tolerant system. It is enough to have $\lfloor 2N/3 \rfloor + 1$ of non-faulty nodes for the chain state to be correct.

The committee of validators itself is a permissioned setup in a sense the nodes can't decide by themselves to become a validator. Instead, all nodes in the committee of validators are bound together via a decentralized DKG procedure. It essentially means that each validator node has to be "co-opted" into the committee by the other nodes. So, this is kind of an agreement between the validators. The main question is, **how trusted can this agreement be?**

In an ISC chain, $\lfloor N/3 \rfloor$ of faulty (misbehaving) validators can stop the chain from producing state updates (it won't produce a wrong result in this case). To produce a wrong state update, $\lfloor 2N/3 \rfloor + 1$ validators must collude.

The chances of $\lfloor 2N/3 \rfloor + 1$ of validators colluding depends on many factors, which all depend on the particular setup of the committee. Here we will present several examples of use case-specific approaches.

5.2 Consortium

A common setup in many corporate use cases for a distributed ledger is a **consortium**, which forms a committee of validators. Consortium partners do not usually trust each other, they are legally bound and they have their real-world reputations. The less consortium partners trust each other the better because misbehavior by any individual party is provable in a deterministic system and competitors are constantly monitoring each other. So, the less trust consortium partners in the committee of validators place in each other, the more secure the chain.

A common reason to form a consortium is exactly because fiercely competing parties do not trust each other yet need to establish a common ground to transact between

themselves. So, consortium partners may leave the consortium and others may join, but collusion among a consortium of competitors is highly unlikely. This makes a consortium of unrelated parties as a whole a highly trusted setup.

Most corporate use cases for distributed ledgers (for example in energy, logistics or financial services), are consortiums of industry market players, (competing) companies and regulators. The public sector examples of consortiums can be treaties like the Schengen treaty or the European Union itself.

Similarly, a consortium of competing crypto exchanges running an ISC chain or chains with a Decentralized Exchange (DEX) in a consortium setup will be way more trusted than any centralized exchange, run by one entity.

5.3 Corporate or DAO setup

An organization may want to distribute a system among its departments. The fact that validator nodes are run by different departments across the globe would make a fault-tolerant system trusted by the organization's headquarters (which do not *a priori* trust its departments) and by the clients of the organization (which trust the organization to a certain extent anyway).

A corporation may want to outsource the validation of its core chains to a distributed set of external validators, by asking each of them to stake funds as collateral to protect against misbehavior. The hiring of external validators may take form of bidding in an open market, essentially a permissionless setup. The setup may also include elements of DAO, for example voting on certain decisions. An example of such a setup would be the Binance Smart Chain[11].

A distributed autonomous organization (DAO) is essentially a corporate setup, governed by a community. It may want to outsource validation of chains to other distributed parties just like any other organization, by asking for security bonds to prevent misbehavior.

5.4 Shared security: multiple chains with shared validators

Let us say the same validators validate multiple ISC chains. The same set of validators form a committee for each chain. In this configuration, all these **multiple chains will have the same source of trust**, and therefore **shared security**. The trustless transactions between those chains remain inside the same trust assumptions regardless of the particular chain. It makes this kind of multiple chain system one sharded distributed ledger with many parallel shards, a highly scalable system. It will run **parallel yet interoperable smart contracts** on it.

This configuration may be used for a consortium or corporate setup. For example, the same DAO could be running separate parallel chains for different functions: tokenization, DEX, oracles, other dApps and so on. The scalability of such configuration is practically unlimited.

[11] Binance Smart Chain Validator: https://docs.binance.org/smart-chain/validator/overview.html

5.5 Shared security: a permissionless setup

By **permissionless setup**, we understand the ability for **any** entity to come and claim its role as a validator for a chain in a open and transparent market for validators.

For a permissionless setup for ISC chains, we envision security architectures similar to those used in Ethereum optimistic rollups, Polkadot, BSC and so on, to be implemented using ISC chains as a base protocol. The following elements are essential:

- Each validator comes with a security bond, an asset which is put as a collateral to a third trusted party, usually a (staking) smart contract.
- Any other third party, called *fisherman*, can provide **fraud proof**, a cryptographic proof that the committee of validators has produced a wrong update of the chain's state. The fisherman can be motivated to monitor the activity of a chain by a bounty and will be discouraged from not being alert by periodic unexpected checks.
- Chain validators already have the valid chain's state, and calculate the next valid state, so they are prime candidates to act as fishermen. They are incentivized to submit fraud proofs of invalid state transitions to collect the stake from other validators and secure their own chain - this means that a chain is secure from malicious state transitions as long as there is at least 1 honest validator in the committee of validators.
- The submitted *fraud proof* will be validated by the staking contracts. This is possible due to the determinism of VM and signatures. If the claim is valid and proves the state update is invalid, the staking contract punishes the committee of validators by taking its security bonds (also known as "slashing").

This setup ensures game-theoretical security of the chain validation: the assumption is such that no party will misbehave as long as the net gain from misbehaving (in this case, stealing funds) is negative. Obviously, security of the chain depends how big the security bonds are.

The description above is significantly generalized and omits details such as the exact form of fraud proofs. Also, certain synchronicity assumptions must be made, for example a time window for submitting proofs and the Dispute Time Delay (DTD).

The essential element of the permissionless setup is the trusted third party, a staking smart contract, which keeps security bonds and handles the fraud proofs (also known as *dispute resolution*). We may see the staking contract as a metaphor for the supreme court. The presence of such "supreme court" common to all chains creates **shared security** among chains that are validated by validators staked in the common staking contacts: the top trust anchor will be the same for all chains. The setup with the "supreme court" creates a basis for a **permissionless market for chain validators**. Any entity can participate in the market with its stake and the chains will hire validators by paying them validation rewards, for example through a fee mechanism and/or interest on the staked capital.

In Ethereum rollups, the "supreme court" is a smart contract run on the Ethereum main chain. The question is, where do the "supreme court" contracts run for the permissionless setup of ISC chains?

The approach of ISC to the permissionless setup is to implement all necessary "supreme court" logic in smart contracts on a **root chain**, which is a singled-out ISC chain. The root chain would be a shared source of trust and security for all ISC chains validated in the permissionless setup. It will run all staking logic as well as open validator market smart contracts.

The question then arises, what entities will validate the *root chain* itself?

In the IOTA 2.0 ledger, the validators of the root chain will be so-called *high mana nodes*. *Access and consensus mana* is a scarce resource produced by active value transfers on the ledger. The high mana nodes are the validators which have the biggest influence and skin-in-the game in the validation of L1 ledger. Taken together, high mana nodes, which concentrate the dominating amount of mana in the network, are as trusted as the whole IOTA 2.0 ledger.

Validation of the *root chain* by the *high mana nodes* equals out the security assumptions of the IOTA Smart Contracts chains on L2 with the security assumptions of the IOTA L1 ledger.

In the pre-Coordicide network, run, for example, by a *distributed coordinator*, the root chain will be validated by the same entities that run the distributed coordinator. This creates a shared source of trust for L1 and multiple L2 ledgers.

Note that the IOTA Foundation's strategy is to get rid of the coordination of the network in the public mainnet. At the same time, the configuration of the L1 ledger with the distributed coordinator and the root chain as a system-wide source of trust and shared security is a powerful platform for big corporate and public networks of pan-European and global scale. So, the coordinated configuration of the IOTA L1 will most likely be kept as a supported version and an option for huge corporate deployments even after getting rid of coordinator in the public mainnet.

6 The IOTA Smart Contract chain

6.1 Smart contract chains

ISC is a multi-chain environment. We can run many parallel blockchains (*chains*) on top of the IOTA ledger, on L2. Each chain has the following properties:

- It has its own ledger state which can be **updated independently** from states of other chains, in parallel.
- The ledger state of the chain uses an **account-based model** of the ledger (contrary to the UTXO model of the L1).
- Its state is **committed** (or **anchored**) in the specific IOTA UTXO ledger account on L1 (see State. Anchoring the state).
- The chain is validated by its own set of validators, known as the **committee of validators**. The validators reach consensus on the state update via a BTF consensus

protocol (see Consensus). Then validators provide the **proof of consensus** to L1 ledger as part of the anchoring process;

- Each chain hosts multiple smart contracts. Within the boundaries of the chain, they are fully composable via synchronous calls, just like smart contracts on any blockchain. We call it **synchronous composability** (also known as **atomic composability**).
- Each smart contract can transact (i.e. exchange assets) with other smart contracts on other IOTA Smart Contracts chains in a trustless and feeless manner through the anchoring mechanism on the L1 ledger. It makes smart contracts fully composable cross-chain. We call it **asynchronous composability**.

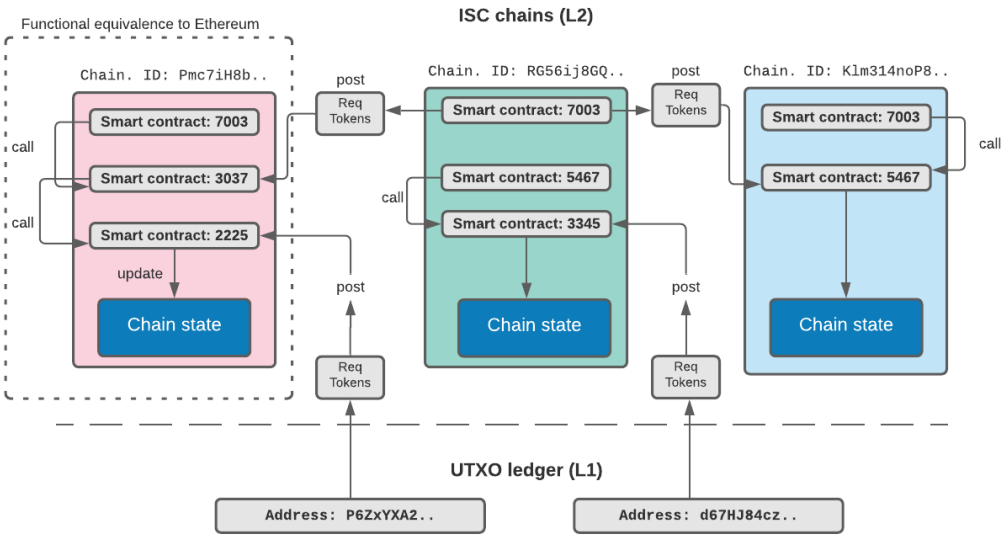

Figure 2: Multiple chains with multiple smart contracts

6.2 Chain account and Chain ID.

For details on UTXOs and the L1 ledger, see Annex: Assumptions about L1 UTXO ledger

A chain account is an account on the UTXO ledger (L1) controlled by a **chain address** also known as **chain ID**. It is a special kind of address, an *alias address*, which abstracts the controlling entity (the *state controller address*) from the identity of the chain: the controlling entity of the chain may change, while the *chain ID* stays the same transparently for user of the ledger.

The chain account contains many UTXOs, including the *state anchor* (also known as the *alias output*), an on-ledger backlog of requests and other L1 assets.

The *chain ID* is the unique identifier of the chain for its lifetime. It is assigned (or *minted*) by the protocol upon creation of the first alias output in the origin transaction of the chain and remains immutable throughout the lifetime of the alias output. The chain

on the UTXO ledger is uniquely identified through its alias address. The protocol guarantees that the **unique alias output** with particular chain ID (*alias address*) is always present in the L1 ledger state.

The ISC chain controls a collection of UTXOs in its account. All UTXOs on the chain's account belong to one of two groups:

On-chain assets are UTXOs that are tracked by the chain in its data state (see On-chain accounts). These UTXOs represent L1 assets owned by the chain on behalf of its smart contracts. The on-chain assets may be all types of UTXOs: extended outputs, NFTs, foundries, alias outputs and, potentially, others. The collection of on-chain assets is therefore committed in the the anchor (alias output type) in the chain's account itself. The immutable anchoring together with the data state makes the on-chain assets a deterministic (objective) part of the chain's account: every validator node perceives the on-chain assets the same way.

Backlog of on-chain requests are UTXOs in the account that were not yet processed by the chain. They are added to the chain's account by asynchronous agents (also known as the requesters) in parallel. It is the non-deterministic part of the chain's account, because validator nodes may have different perception of it each moment of time. The chain is processing the UTXOs (also know as On-ledger requests) in the backlog by consuming UTXOs and thus removing them from the backlog. Processing the backlog is an **atomic operation**: removing the UTXO from the non-deterministic backlog and adding the effects of its processing into the deterministic data state.

See also Figure 5: "Balance sheet" on L1 and L2.

6.3 State. Anchoring the state

The state of the chain is deterministic, i.e. perceived equally by all validator nodes. It consists of:

- **On-chain assets** on L1 (see above).
- a collection of arbitrary key/value pairs (known as the **data state**) on L2. The *data state* which represents internal chain accounting as well as the use-case-specific data stored in the chain by its smart contracts. The data state is stored in their databases by all validator and access nodes of the chain (see Nodes).

By **anchoring the state** we mean the following:

- Accounting of all on-chain assets in the data state, thus tracking which on-chain asset or balance belongs to which smart contract or L1 address.
- Placing the **commitment to the data state** and the **state index** into the alias output, always contained by the chain's account.

  The *commitment to the data* is a value, such as a root of the Merkle tree of the data state, which makes it impossible to modify the data state in the state without changing the commitments. See Committing to the state.

The *state index* (also known as *block hight*) is an incremental value which incremented with each new state anchor. The consistency of incrementing of the *state index* is enforced by the UTXO ledger on L1.

Anchoring guarantees an atomic relation between the data state (through commitment) and the assets owned by smart contracts.

The *alias output* of the chain guarantees there's always **exactly one** such output for each chain on the UTXO ledger. We call it the **state anchor** and the containing transaction the **anchor transaction** of the chain.

With the anchoring mechanism, the UTXO ledger on L1:

- Guarantees **global consensus** on the state of the chain. There is no possibility to fork the chain.
- Makes the state **immutable** and tamper-proof.
- Makes the **state transitions controllable, atomic and consistent with the asset balances**.
- Makes the data state **global and objective** within the chain: the requirement for quasi-Turing complete smart contracts.

The state anchor contains:

- The identity of the chain (the *chain ID*, an *alias address*).
- The address of the *state controller* (the address with the private key behind to control the alias).
- A commitment to the data state (see Committing to the state).
- The state index (also known as block height), which is incremented with each next state output, the state transition (see The state transitions).
- A consistent timestamp of the state.

The state commitment and the on-chain assets are owned by the chain's account. The anchor output and all assets controlled by the alias address can only be moved by providing the valid signature of the state controller, which itself is specified in the alias output.

For state controller signatures, ISC uses threshold signatures (BLS or based on Schnorr/ED25519). It ensures that at least $\lfloor 2N/3 \rfloor + 1$ validator nodes have to provide valid signatures for the anchor transaction to make it valid.

This way, the chain's state and assets are controlled by the **quorum** of validators in the committee of validators.

The committee of validators can be **rotated** (changed) by changing the state controller address in the state anchor. Note that rotation won't change the identity of the chain, the chain's account, controlled assets, data state and backlog of on-ledger requests. The rotation of the state controller is completely transparent to smart contracts and the chain as such.

6.4 Committing to the state

The *data state* of the chain is a collection of key/value pairs. Each key and each value are arbitrary byte arrays. The key/value pairs of the data state are used to implement all kinds of on-chain concepts: from On-chain accounts to use case-specific data of the smart contract state.

The ISC concept of the chain's state differs a bit from the same concept in blockchains: the chain's state in ISC is a generic collection of key/value pair, essentially state variables and its values. Unlike blockchains, the chain's state structure does not reflect the history of mutations, i.e. the way how the state was created. The history of mutations (blocks) are kept in the database separately, for auditability and synchronizations reason, however the state itself does not have a concept of block.

In its persistent form, the *data state* is stored in a key/value database outside of the L1 UTXO ledger in the validator nodes of the chain. This means the data state of the chain is within a known circle of nodes. This fact may be used in use cases that seek privacy for the chain data while using the public IOTA network for L1 as well as compliance with European GDPR regulations.

By "commitment to the (data) state" we mean a cryptographical commitment to the data in the state as a whole.

To provide consistent historical information for auditing and other purposes, the chain keeps it in the state itself, namely in the *blocklog* partition of the state. The *blocklog* contains commitments to all previous blocks and states, processed request and events. Therefore, commitment to the data state is always a commitment to its history too.

The most common example of a commitment to the state is the root of the Merkle tree, calculated from the data of the state. In Ethereum, state commitments are computed in the form of Merkle Patricia trie. Merkle trees are not the only way of committing to the state and not the most efficient one.

In the ISC chains, we follow a more efficient pattern of state commitments by using a so-called **verkle tree**: a tree based on vector commitments made by the tree nodes to its child nodes. The vector commitments are usually based on different cryptography than the hash function used in the Merkle trees.

The commitments in the form of tree provide **proofs of inclusion** of data elements (also known as **proofs of existence**), based on the path from the tree root to the data element. Size of the *proof of inclusion* depends on the method of commitments and the branching factor d of the tree (number of children). For hashing-based binary Merkle trees branching factor is $d = 2$, for *Patricia trie* used in the Ethereum it is $d = 16$. The proof of inclusion for the data element in Merkle trees is $d \times O(\log_d N)$ where N is size of the data state (number of key/value pairs in it). This makes Merkle trees not practical[12] for larger branching factors d.

[12] A Theory of Ethereum State Size Management: https://hackmd.io/@vbuterin/state_size_management

By using other types of commitments in the verkle tree, we can make size of the proof of inclusion $O(log_d N)$, i.e. d times smaller. It enables usage of trees with a large branching factor, for example 256. That makes state proofs very fast, small and efficient. The most promising commitment schemes for *verkle trees* are *Kate (KZG)* and *Pedersen* polynomial commitment schemes.

Let's say we have a state of the chain S. In parallel to the key/value storage of S, the ISC nodes maintain verkle tree $trie(S)$. The commitment to the state $C(S)$ is the root of the verkle tree. It is stored in the state anchor on L1 in order to make the state of the chain S immutable to anybody but the mutations produced collectively by the validators of the chain. Note that the state contains history of itself in the *blocklog* partition, therefore the chain is commited to the history too.

The state of the chain is updated by applying a mutation M_i to the current state S_i: $S_{i+1} = apply(S_i, M_i)$. Mutation M_i is a state transition, calculated by the virtual machine from inputs and the previous state: $M_i = VM(S_i, inputs)$. The *inputs* are the requests to be processed by the VM, while the smart contract programs are part of the state S_i.

The important requirement to the state commitment scheme is the ability to efficiently update the verkle tree from the mutation: $trie(S_{i+1}) = updateTrie(trie(S_i), M_i)$.

In ISC, the data state of the chain is stored in its entirety, as a current state (see also The state transitions). To access a key/value pair in ISC you do not need to prove its inclusion in a certain block by traversing through the Merkle tree, like in Ethereum and Bitcoin blockchains. The blocks (which are mutations to the state) are only kept for enabling unsynced nodes to catch up and for auditability purposes. So, a tree of state commitments is not needed for the operation of the chain.

However, the state commitments are needed in the following use-cases:

- Committing to the state as a whole, sub-states and the auditable history in the UTXO ledger state on L1.
- Proving inclusion (existence) of a key/value pair in a state of almost arbitrary size in distributed long term storage solutions.
- Statelessness: ability to operate consistently on a fragment of the state and proofs of inclusion of that fragment. This property is very important for fraud proofs and shared security solutions.
- State pruning: ability to delete historically old data from the state without losing the ability to prove inclusion of any pruned data element in the state. This is a prerequisite for any hight TPS use case when databases with chain state quickly become too large.
- Handling European GDPR requirements to "forget" certain data: data can be deleted from the state without violating the consistency of the state.
- Snapshotting the state: storing commitments to the past states in the state itself. This enables the validation of a past state with reference to the current state.

6.5 The state transitions

The data state is updated by mutations of its key/value pairs. Each mutation is either setting a value of the key or deleting the key with the value from the data state. Any update to the data state can be reduced to an ordered sequence of mutations. The state is updated (mutated) by applying the mutations in sequence to the current data state.

A set of mutations to the data state which is applied as a single unit we call a **block**:

$$nextDataState = apply(currentDataState, block)$$

The **state transition** in the chain occurs atomically together with the movement of

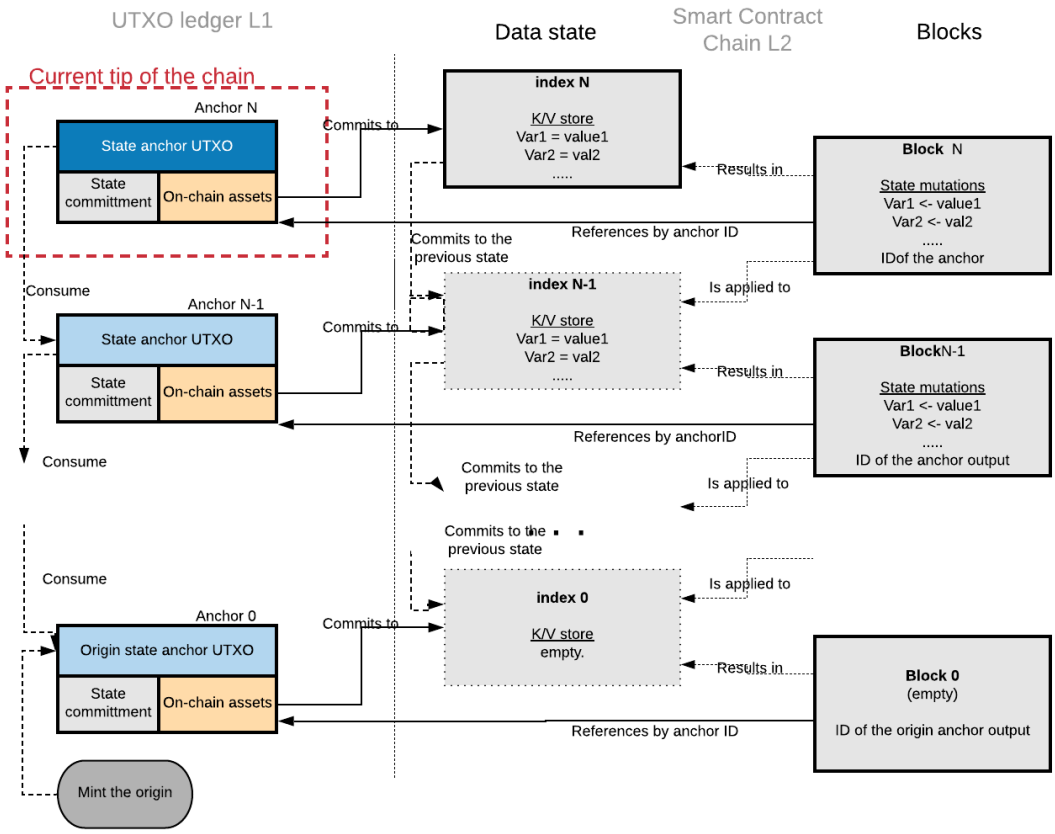

Figure 3: Chain on L1 and L2

the chain's assets and the update of the state commitment of the new data state in the transaction which consumes the previous state output (or 'previous state anchor') and produces the next state anchor.

At any moment in time, the data state of the chain is a result of applying the historical sequence of blocks, starting from the empty data state. Hence blockchain. The ISC block does not contain block headers nor transactions, only mutations of key/value pairs.

In parallel, the history of the state is represented on the UTXO ledger on L1 as a sequence (or 'chain') of state anchors, each controlling the chain's assets in the chain's L1

account and the commitment to the data state. See figure Chain on L1 and L2.

Note that not all of the state anchors back in the past may be available: due to practical reasons the older transactions on L1 may be pruned (deleted) and made unavailable. It is guaranteed, however, that the tip of the chain of anchors UTXOs is always available in the ledger state, it is globally unique and it can easily be located in the chain's account by the chain ID.

The blocks and state outputs that anchor the state are computed by the VM, which is a deterministic processor or a "black box". The VM is responsible for the consistency of state transitions and the data state.

6.6 State synchronization and validation

Any party (a node in particular) with access to the chain of blocks and the UTXO ledger on L1 can reconstruct the data state of the chain by sequentially applying blocks to the origin state or starting from some valid snapshot of the state.

The node can validate the state by computing commitment to it and comparing it to the state commitment on the state anchor on the UTXO ledger state in L1: state is valid if value on the state anchor is equal to the computed commitment.

Each node keeps its own copy of the data state in the database and checking with the state anchor guarantees that it is the only possible valid data state.

The process of keeping the copy of the state valid and up-to-date (i.e. the latest one) we call **state synchronization** or **syncing**. The latest (the current) state is determined by the unique UTXO of the state anchor.

The party can query the UTXO ledger at any time and retrieve the current state anchor of the chain. If the *state index* and *state commitment* contained in the state anchor is equal to the state index and the state commitment of the current data state in the database, it means the party has the up-to-date state in its database, i.e. it is synced.

If the state index on the data state stored in the database is less than the *state index* of the state output, it means the node is behind, i.e. not synced. Note that from state index the node knows how many blocks exactly it is behind the latest state.

The node can catch up and synchronize its state by the iterative process of querying missing blocks from other nodes and applying them in sequence to its own data state in the database. Each such step produces new data state a the node can calculate commitment value to it. The process is repeated until the data state reaches a state index and state commitment value equal to the commitment in the current state anchor. Note that:

- Older state anchors may not be available in the ledger (in other words, it is 'pruned'). However, the process will always finish because the current state anchor is always on the ledger state.
- The process may start from the origin state. This is not practical in many situations, so we may want to start syncing from a snapshot of the data state. In this case,

we will find out if the snapshot represents a valid past state by reaching the current state (or not).

6.7 On-chain ledger

Each ISC chain implements its own ledger with its own validity constraints in its data state. The deterministic logic of the The virtual machine (VM) is responsible for keeping the on-chain ledger consistent. Chain ledger state transitions are computed by the VM and nothing else.

6.7.1 Smart contracts

A smart contract is a program deployed on the on-chain ledger. By 'deploying' we mean the smart contract has a binary code stored in the data state of the chain, which makes it immutable. The binary code is interpreted by the deterministic VM. It makes each smart contract a deterministic extension of the VM.

Each smart contract (or 'program') has access to a partition of the chain ledger. Only a particular smart contract has read/write access to its partition. The entire data state of the chain is split into non-intersecting partitions or sub-states, each controlled by a smart contract.

Some partitions, known as 'system partitions' are built into each chain . Each system partition is controlled by the corresponding **core smart contract**. The Core contracts are hardcoded into the VM and are deployed automatically. They are responsible for the on-chain infrastructure used by user-defined smart contracts, such as a registry of smart contracts, on-ledger accounts or chain configuration data. Core smart contracts may be seen as a part of the VM itself.

6.7.2 On-chain accounts

The chain ledger implements its own ledger of accounts, the **on-chain accounts**. The ISC chain uses the account-based model of the ledger (unlike the UTXO model used by L1). Accounts are global for smart contracts within the chain. Smart contracts can assume a shared and objective global state of accounts.

The ledger of accounts is implemented as a partition of the data state of the chain, controlled by a core smart contract called ***accounts*** contract. Each on-chain account contains:

- Balances of iotas and other digital assets (such as foundries or NFTs) native to L1.
- The account owner, who is the controlling entity of the account. Only the owner can move assets from the on-chain account. The account holds the ID of the owner as *accountID*.

The *accountID* is either an address on the L1 ledger (with the private/public key pair behind it) or an ID of a smart contract, on the same as another ISC chain. The ***accounts*** contract provides controlled and secure access to funds by its owners. The sender of the request to access the account is always securely identified through the signature (see Requesting smart contracts).

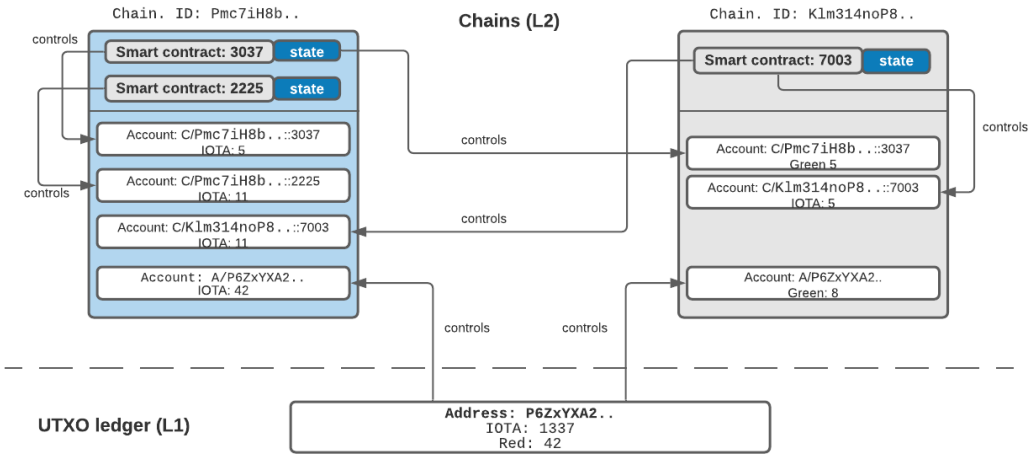

Figure 4: On-chain accounts

The consistency between L1 on-chain assets controlled by the chain and on-chain ac-

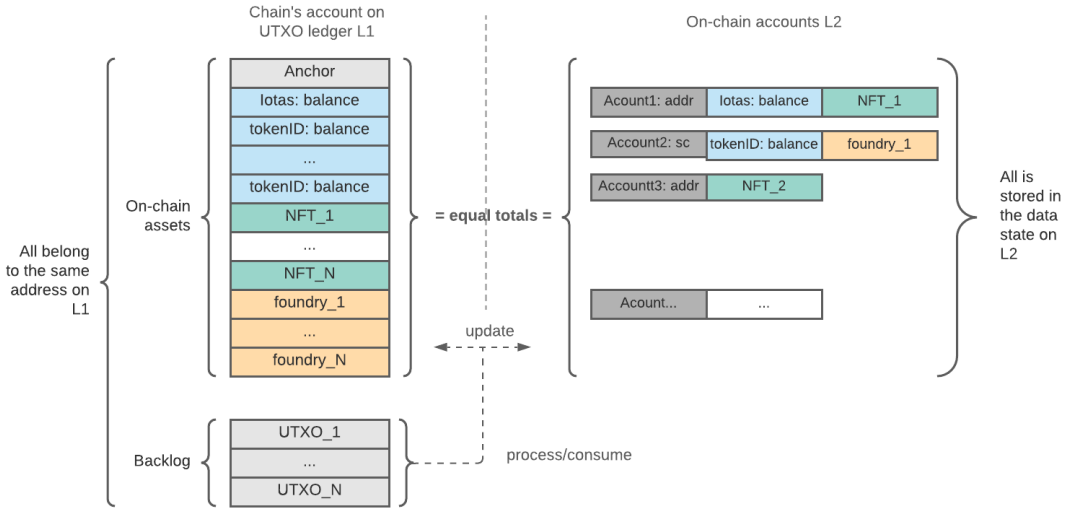

Figure 5: "Balance sheet" on L1 and L2

counts on L2 chain is guaranteed by:

- atomic updates of the data state and L1 account via state anchoring mechanism;
- the determinism of the VM in general and the ***accounts*** core smart contract in particular.
- the consensus of validators.

The VM logic guarantees the main invariant of the on-chain account ledger: any state transition keeps the sum of assets in the on-chain accounts equal to the assets controlled by the on-chain assets on L1. The above is equivalent to the equal assets and liabilities

on the balance sheet in conventional double-entry accounting: on-chain accounts may be treated as balance sheet liability side and on-chain assets on L1 are balance sheet asset side.

The *backlog* has semantics of "not yet/to be processed inputs/assets". See Requesting smart contracts.

7 Nodes

7.1 Validator nodes. Access nodes

Each chain has a **subnet** of ISC nodes. We call the nodes in the subnet **access nodes**. By calling it this way we want to say that (a) each access node holds a valid state in its database and (b) it (optionally) provides access to smart contracts for external callers which want to query the state of the chain via view calls and want to send off-ledger requests directly to the access node.

Each access node is connected to the IOTA L1 node in order to receive updates on the chain's account. See also Incentives to validate L1.

The peers in the subnet (in other words, other access nodes) are connected to each other through statically assigned trust relationships. This means that each access node has its neighbors in the subnet with known identities. In general, not all access nodes are neighbors of all others, so the topology of the subnet may not be homogenous.

Some access nodes in the chain's subnet are **validator nodes**. The validator nodes form the **committee of validators**, which is an ordered set of nodes. All validator nodes on the committee are trusted neighbors of each other.

Validator nodes may or may not provide access via API to the outside world. It is common for validator nodes to be exposed inside the subnet but not to the outside world: the other access nodes may shield them to protect from external attacks.

The committee of validators collectively runs the **consensus** on inputs and outputs of the VM calculations, produces the state update and the anchor transaction (see Consensus).

Other access nodes are listening to new state anchors submitted by the committee of validators to the IOTA L1. Whenever a new state anchor is detected, the access node synchronizes its chain's state with other nodes by requesting blocks from other nodes and validating the state against the new state anchor.

The validator nodes usually each is having access to the L1 via own IOTA node. This makes access to the Tangle on L1 as distributed as the chain's validators themselves: no single point of trust for access to the L1 ledger.

Each validator node of a chain has a private key share, known only to it. With its private key share, the validator node signs the transaction. The consensus process collects signatures and aggregates them into the final valid threshold signature without needing to access the master secret, not known to anybody. It is enough to have a part of signatures

T to produce a valid final signature. Usually, threshold parameter $T = \lfloor 2N/3 \rfloor + 1$ where N is the size of the committee of validators.

The private key shares are distributed among validator nodes during the secure *Distributed Key Generation (DKG)* procedure (in the first versions of ISC we are using Rabin-Genaro algorithms[13]. With the rapid development of the field, new interesting alternatives are coming).

Each collection of keys represents a particular committee of validators identified through the master public key and the *state controller address*. The *state controller address* is contained in the *alias output* of the state anchor transaction. That gives control over the state and the funds of the chain to the quorum of the validator nodes: the L1 ledger requires proof of consensus among the quorum majority of validators on the state transition; otherwise a state transition is not possible. The threshold signature is the proof of consensus.

The committee of validators can be changed (rotated) by replacing the state controller address in the state anchor output. This way the committee of validators can be modified by adding, removing or replacing nodes without changing the chains' identity and is completely transparent to the smart contract and its requestors.

7.2 Consensus

The state update of the smart contract chain is a result of a consensus procedure among its validator nodes. The consensus used in ISC is leaderless and asynchronous. We describe it here in general terms:

- Upon the start of the new cycle (which is known from L1 as a shared source of truth for nodes), each validator node queries its mempool and produces a batch of requests, known as the **batch proposal**. It is very likely that batch proposals will be different for each validator node, because mempools are asynchronously filled with on-ledger and off-ledger requests. Each validator node puts its subjective **timestamp** and signature by its own private key share of the current state hash into the batch proposal.
- Validator nodes run a distributed consensus algorithm to reach a consensus agreement on the batch of requests based on inputs, the batch proposals. ISC uses the **HoneyBadger Asynchronous Common Set (ACS)** algorithm for this: a leaderless asynchronous Byzantine Fault Tolerant consensus protocol. The absence of time assumptions (due to asynchronicity) and absence of leaders makes consensus resistant to certain attacks.
- The output of the ACS is used to deterministically calculate a **list of requests** the same in each validator node, the **timestamp** and the **unpredictable randomness**. The timestamp is calculated deterministically from the set of proposed timestamps and adjusted to the time constraints of the state. The randomness is taken from the aggregated threshold signature of partial signatures provided as part of the input to the ACS consensus.

[13] Secure Distributed Key Generation for Discrete-Log Based Cryptosystems: https://link.springer.com/content/pdf/10.1007/s00145-006-0347-3.pdf

- The batch of requests is ordered deterministically based on the unpredictable randomness generated by the ACS consensus. This way, the order of requests in the final batch of requests is unpredictable for any validator node and therefore cannot be influenced by less than a quorum of validator nodes. The randomization of the batch is a way to prevent the *miner extractable value* (MEV) insider attacks such as *frontrunning* or *sandwiching*.
- The consensus guarantees exactly the same batch of requests, timestamps and unpredictable randomness in each validator node. All requests in the final batch can be processed by the same quorum of validator nodes, i.e. each validator node has enough information to calculate the result by itself, independently from others.
- If consensus on the batch of requests is not empty, it is used as an input for the VM. Otherwise, the round is restarted with a new batch proposal.
- Deterministic VM produces as the output: (a) mutations for the state (the block) and (b) a state anchor transaction essence (not signed yet). All non-faulty nodes produce exactly the same output, the only possible valid result. Different results mean the Byzantine fault, which is a misbehavior.
- Each node signs the transaction it calculated with its own private key share and disseminates the partial signature to its validator peers (only signature, not the transaction). Note that signed faulty result would be proof of misbehaviour traceable to the specific faulty node.
- Each node aggregates partial signatures received from peers into the final threshold signature and thus each non-faulty node produces identical final transactions ready to be posted to the L1 ledger.
- All nodes posts final transaction in a random but deterministic order to the L1 node. In case several nodes post the same transaction to L1, the L1 consensus takes care of it and ensures the consistency of the UTXO ledger. In the ordinary course of operations, this is a rare event.

After the state anchor transaction is confirmed on the L1 ledger, nodes receive notifications independently from their corresponding L1 node. After the node receives a confirmed anchor transaction it commits the updated state and the corresponding block to the database.

The access nodes which are not participating in the consensus will also receive the new anchor transaction from their L1 node. The access nodes will query the missing block from other nodes, then will validate and sync its state to the latest one.

8 The virtual machine

The **virtual machine** (the VM) is a deterministic part of the ISC chain responsible for consistent state transitions of the on-chain ledger. The VM has a hardcoded part that includes core smart contracts and built-in interpreters such as WebAssembly (Wasm) interpreter or EVM. Smart contracts are extensions of the VM and the deterministic state transition function.

The VM of ISC follows the model of Ethereum VM (EVM): smart contracts are interpretable binaries which can be invoked on-chain by the proposed transaction (this is called

'a request') or from another smart contract. It also can be invoked to access the state of the chain in read-only mode (or 'views'). ISC also makes an effort to abstract core functions of the VM from the specifics of its implementations. See also Invoking smart contracts. Composability.

Upon deployment of the smart contract, its binary code is stored in the data state of the chain. Smart contracts are loaded from the data state and interpreted by built-in interpreters of the VM.

8.1 VM abstraction

Through *VM abstraction*, the VM can support several types of interpreters of smart contract programs at the same time on the same chain: for example, different interpreters of WebAssembly binaries and different configurations of EVM. The core of the chain and the VM is agnostic about the exact type of the binary interpreter of smart contract programs. The VM abstraction includes:

- The abstraction of VM interpreter. The main targets are WebAssembly and EVM.
- The abstraction of the *Processor*, a smart contract program loaded into the interpreter (linked with *Sandbox*) and ready for running.
- The *Sandbox* interface: an abstract interface provided by the VM for its plugin, which is the smart contract. Sandbox provides deterministic access by the smart contract to the resources of the chain: the data state, assets/accounts, calling other smart contracts, posting cross-chain requests, utility functions for heavy cryptography and similar.
- *SandboxView* interface, for read-only access to the chain's state from the read-only methods of smart contracts, which are the view entry points.

8.2 Core contracts

Each chain can have many smart contracts. Some of them, known as the **core contracts**, have special functions in the VM and are hardcoded. Core contracts are deployed automatically as a part of the deployment process of the chain and comply with the same interfaces as all other smart contracts. That makes core contracts invocable by external users and smart contracts just like any other. See also Invoking smart contracts. Composability

Each chain contains six core smart contracts, which are deployed automatically along with the chain itself. Each core contract controls its own partition with chain level data which is maintained by the VM:

- ***root*** contract: deployment of new smart contracts and the registry of contracts.
- ***accounts*** contract: registry of on-chain accounts, ownership, moving assets between on-chain accounts.
- ***blob*** contract: registry of big binary data objects, such as Wasm binaries, associated data schemas, program sources.

- ***blocklog*** contract: keeps track of blocks, receipts of requests and event log. It is the auditable history of the chain.
- **governance** contract: configuration of the chain, such as admin rights, fee policy, committee rotation, contract deployment policy.
- ***_default*** contract: always invoked when the target contract is not found.

8.3 Gas and fees

8.3.1 Gas considerations

The IOTA DLT is feeless at its core, as are the smart contracts in ISC. However, for quasi-Turing complete computations we need the concept of gas to restrict abuse of the processor and storage resources, to prevent runaway programs and the generating of garbage in the state. The concept of gas assumes the cost in gas units to be proportional to the amount of resources consumed.

The purpose of gas is twofold:

- To put a real **price on resources**. It translates into iotas per gas rate. The requester pays iotas for the resources of the chain.
- To put a cap on an invocation of the smart contract, the **gas budget**. Smart Contract programs burn gas when running. Exceeding the budget means cancelling the processing.

Unlike EVM, ISC can have different program interpreters and different gas metering on the same chain. It makes it difficult to calibrate gas costs objectively.

Therefore, in ISC the gas metering is use case-specific and depends on gas and fee policies set by the governance of the chain. Examples of possible gas policies include:

- **No fees, no gas budgets** (it may be acceptable in private setups but easily attackable in public ones).
- **No fees, fixed gas budget** per contract (for corporate and private deployments).
- **Fixed fees** on chain level and per contract, fixed gas budget per contract (mostly for corporate deployments).
- **Gas based fees**, iota per gas price at the chain level, iota-based gas budgets (for public deployments).

Currently, we see gas price in iotas mostly fixed by the governance decision. A gas market is possible and might be a future option; however, volatility in iotas/gas prices in general is to be avoided.

Note that the purpose of the multi-chain environment is to prevent congestion due to the limited throughput of one chain and ever-growing demand. Spawning another parallel chain to increase the total throughput to prevent congestion is always an option in ISC.

8.3.2 Fees

Apart from congestion control through gas as noted above, fees have the function of motivating validators and chain owners (governors) to run and validate the chain.

Usually, fees are paid in iotas or in any IOTA native token. The choice is up to the governance decision. The charged fee (if any) is split into two parts: the **validator's fee** and the **owner's fee**. The splitting policy is set by the governance of the chain.

The validator's fee goes to the account of validators in the committee of validators, usually in round-robin or deterministic pseudo-random fashion. The validator fees are an incentive for the validators to validate the chain.

The owner's fee goes to the so-called **common account** on the chain, controlled by the *governor* of the chain (see below).

9 Governance of the chain

ISC is a multi-chain environment where smart contracts on different chains can transact without any additional trust. At the same time, each ISC chain is a separate distributed ledger with specific parameters and configurations.

Examples of use-case specific chain parameters and configuration include:

- The governor itself.
- Parameters for rotation of the committee of validators.
- Configuration of the subnetwork of trusted access nodes of the chain.
- Fee policy. Gas rates. It defines if and how the fee is calculated, charged, distributed, in which kind of tokens, fallback mechanisms, etc.
- Permissions (if restricted) to deploy smart contracts, to use blob space, to post requests, to set all kind of limits, etc.
- Numerous technical parameters and limits for consensus and smart contract execution.

The entity authorized to change the configuration of the chain is called **the governor** or **the owner**. The core smart contracts check authorization for restricted operations.

The governor can only do what is prescribed by the core smart contract; it cannot do anything to change the deterministic flow of chain updates on the chain. For example, the governor does not do any transaction validation; that is the job of the validator nodes. Similarly, the validator nodes are not involved into any decisions on the chain's parameters.

The governor of the chain is represented by one of two options:

- An ordinary IOTA **address** with a public/private key pair behind it. Usually this is a **centralized** model of governance where one person or organization determines parameters of the chain.

- A **smart contract** on the same or another chain that performs governance operations on the chain on behalf of a distributed governing body – for example, a DAO. This is a **decentralized** model of governance. The governing smart contract may implement its own governing logic: for example, it may require secure voting by the governing body on certain decisions. The scope of governance models implemented in the **DAO smart contract** is unlimited.

Examples of functions of the governance contract of the chain include:

- Transferring the ownership (governance rights) to another entity through grant/claim protocol.
- Rotating validators of the chain to another set of validators (committee) represented by the address (this is a multi-step process).
- Setting fee/gas policy. A fee policy also includes how fees are distributed among validators and the owner of the chain.

One special account on the chain, the *common account*, is always controlled by the governor, whoever it is. Any assets/tokens sent to core contracts, intentionally or by mistake, end up in the common account and can be moved only by the chain's governor/owner.

10 Invoking smart contracts. Composability

There are two general ways of invoking the smart contract:

Calling a contract is a **synchronous** invocation of the program.

Requesting a contract is an **asynchronous** invocation of the program.

The two methods have different semantics and they are used in different environments. When used by smart contracts to invoke other smart contracts, these two methods represent two different ways of composability of smart contracts:

- The calling offers a **synchronous composability**(also known as **atomic composability**) of smart contracts on the same ISC chain.
- The requesting offers an **asynchronous composability** of smart contracts cross-chain, i.e. between smart contracts on different chains.

10.1 Calling smart contracts

In the blockchain, smart contracts interact by calling each other, just like functions in the program. This is the main way of interoperability between smart contracts on blockhains like Ethereum. This way, smart contracts on the same chain become **atomically composable**. The *atomicity* points to the fact that all smart contracts access the same global state in the context of one block and modify it in a synchronous manner; therefore, either all updates of all smart contracts are included in the ledger with the block, or none of them are. For example, if a contract calls an ERC-20 contract and then a Decentralized Exchange (DEX) contract, and the DEX calls the oracle contract, all in the same block, then they are all in the same state.

Synchronous composability is an important property of blockchain smart contract

ecosystems, such as Ethereum. ISC fully supports this type of composability inside each smart contract chain. This type of composability has limited scalability due to the global objective state assumption.

A **view call** is a synchronous invocation of a smart contract from off-chain by calling the read-only (view) entry points. The smart contract program is run in the context of the current state of the chain and the result is returned to the caller. The view calls are used for example by the web server to display the state of the smart contract on the web page.

10.2 Requesting smart contracts

A **request** is a way to invoke a smart contact function from the environment which is outside the chain. Requests are sent (or 'posted') by users from their wallets or by other smart contracts on other chains. Each request bears a signature by the private key of the sender and therefore the sender of the request is securely identified. The request 'wraps' a call to a smart contract.

The flow of requests to the smart contract does not have a deterministic order. The requests are asynchronous by their nature. After the request is sent/posted by a wallet user or by a smart contract, it reaches the mempools of the validator nodes of the target chain. It is then picked by the distributed consensus process among validator nodes, and the smart contract program is called (see above) with the call data wrapped in the request on each validator node with the same state as an input. The result is settled as a state update of the chain (and the smart contract) in the new block on L2 and the anchor transaction on L1.

The request to the smart contract contains the following:

- Sender ID (sender address).
- Target chain (chainID, an alias address).
- Target smart contract ID in the chain.
- Target entry point ID in the contract.
- Attached assets: iotas and other native tokens.
- Call parameters: a collection of key/value pairs.

The on-ledger requests also may contain UTXO options: timelock deadline, expiry deadline, return amount and others (see Annex: Assumptions about L1 UTXO ledger).

There are two ways a smart contract can be requested:

- In the form of a UTXO transaction on the L1 UTXO ledger: this is the **on-ledger request**.
- In the form of signed API call data sent directly to the access node of the chain: this is the **off-ledger request**.

10.2.1 On-ledger requests

An on-ledger request is a call to the smart contract wrapped into the UTXO. It means, the on-ledger request is a transaction on L1. The target address of the UTXO is treated as an ID of the target chain.

All UTXOs in the chain's account that are not recognized as an on-chain asset are treated as **on-ledger requests**. They make up a *backlog* of on-ledger requests to be processed (see Chain account and Chain ID).

The chain, a L2 entity, takes unprocessed requests from the backlog, processes them and removes them from the backlog.

The on-ledger requests are used:

- **To send (deposit) assets from user wallets** (addresses) to smart contracts on chains. The wallet creates a transaction with the output which has the target chain ID as target address and other call data is wrapped in the metadata of the output. The output representing the request appears upon confirmation in the backlog of the target chain and is processed.
- For **cross-chain transactions** between smart contracts on different chains. The smart contract posts the request to another contract on another chain by including output with wrapped call data into the anchor transaction of the state update.

The operations of submitting a request to the backlog of the target chain and removing it from there are **atomic operations**. In case of cross-chain requests the two events, (a) state transition of the sender chain and (b) the appearance of the sent request in the target backlog, is an atomic unit.

10.2.2 Asynchronous composability

The on-ledger requests provide a basis for the trustless **asynchronous composability** of smart contracts on different chains via **guaranteed and trustless delivery of assets and data** enforced by the UTXO ledger on L1.

It usually works with the expiry time option in the UTXO (see also Annex: Assumptions about L1 UTXO ledger). The sender is guaranteed that once the request is sent to the target chain, it will be processed by the target smart contract or otherwise the assets (the output in its entirety) will come back untouched. So, L1 guarantees consistency even with asynchronous requests without any additional trust in between chains: there is no need for relays and cross-chain bridges.

Due to the properties described above, by using common standards of the request metadata (which may also be enforced by the UTXO ledger), smart contracts become composable between different chains, even without synchronicity assumptions. We call it **asynchronous composability**: the kind of composability between smart contracts which **does not create scalability bottlenecks**.

For example, one chain can wrap its ERC-20 or similar smart contract asset into the cross-chain request and send it to another chain to sell it on the decentralized exchange

(DEX). The only trust involved in this transaction is the trust to the smart contract chain running the DEX; however, the cross-chain communication is happening on the same ledger, therefore there is no need for additional assumptions about trust in between. The two chains run their smart contracts independently from each other, i.e. in parallel. The wrapped asset is always traceable back to the originator, the ERC-20 contract and the chain.

10.2.3 Off-ledger requests

The off-ledger requests are sent by users from their wallets directly to the access nodes of the chain. It is an API call, not a transaction on L1. They can not be sent by smart contracts to other smart contracts (smart contracts are agnostic about nodes).

The off-ledger requesting is much faster and much less costly than requesting on-ledger (for example it does not require dust deposit). The off-ledger requesting makes high throughput (many hundreds of TPS on one chain) a norm.

Off-ledger requests are delivered to the mempools of validator nodes directly through a dissemination mechanism in the chain's subnet. The off-ledger requests bypass the UTXO ledger, so its off-ledger throughput is limited only by the smart contract chain itself, not by L1.

The off-ledger requests are functionally equivalent to submitting a transaction to an Ethereum node.

The off-ledger request contains the same call data as the on-ledger request, except for UTXO-specific options (e.g. timelock). It always has the signature of the sender and an account-specific **incremental nonce** as part of the replay prevenion.

The signature securely identifies the sender. The VM checks if the sender has an account on-chain with enough balance for fees. If the account does not exist or the balances are not enough for fees, the processing is canceled. The asset balances specified in the request refer to the assets in the sender's on-ledger account. After these checks pass, the off-ledger requests are functionally equivalent and have the same security properties as on-ledger requests.

The on-ledger requests cannot be replayed; this is guaranteed by the uniqueness of the transaction hash. The off-ledger requests, however, requires a special mechanism to prevent replay, i.e. repeating the same signed request by the attacker.

In order to **prevent a replay of off-ledger requests**, the VM checks the receipt of the request by the ID on the data state (*blocklog*) to see if it wasn't already processed in the past. Under the assumption that we have all request IDs of the past and the IDs are unique (never repeats), it is enough to prevent replay. However, due to practical requirements of keeping the database at a reasonable size old request receipts are pruned from time to time in the state. That makes it impossible to determine if the request was already processed in the distant past or not.

To prevent replaying requests from far in the past, the VM checks if the nonce is not

too old for the account. The user is required to send requests by incrementing nonces of their requests one by one. The nonce is a part of the request's hash (ID). The overall replay protection scheme is designed to circumvent the problem of limited historical memory of already processed requests and the absence of deterministic order among nonces in the batch of requests processed in the block. For reasons of space, we will not go into detail here.

10.3 Tokenization. Bank and cash metaphor

The design of the multi-chain system of ISC includes the concept of **advanced tokenization on L1 and L2**, also known as the **tokenization framework** (see Annex: Assumptions about L1 UTXO ledger).

The smart contracts themselves already have powerful tokenization features via contracts equivalent to ERC-20 and similar. In ISC, this is known as **L2 tokenization**. In addition, smart contracts on ISC chains can own and manipulate L1 assets such as iotas, native tokens, NFTs and so on.

The native tokens and special UTXO types, like foundries, make tokenization on L1 possible.

The concept of *advanced tokenization* integrates both L1 and L2 into one framework. We also use the bank and cash metaphor when reasoning about it.

The "bank" would be a smart contract on L2, with its own system of accounts equivalent to ERC-20, for example. These on-chain accounts are IOUs for its users, just like in real banks.

The "cash" would be native tokens on L1, which the "bank" mints for its clients by destroying corresponding IOUs on the chain. It **locks** the corresponding L2 tokens (IOUs) in the smart contract (for example, ERC-20) and hands over the newly minted L1 token to the user´s L1 address. So the user (the address) will stop being able to access those ERC-20 funds on the chain. Instead, the user will own a native L1 token in the wallet and will be able to transact it on L1 with others, just like paying in cash.

The IOTA UTXO ledger supports this *bank and cash metaphor* by the function of *minting* of **native tokens** by Foundry outputs out of thin air. The foundry output is unique on the ledger; it has an immutable ID and enforces a minting schema, which is immutably linked to the Foundry when it is created.

Once minted, the new supply of native tokens is given the token ID and is therefore always traceable to the specific foundry output. The UTXO ledger enforces consistency of the supply, such as the circulating number of tokens, with the immutable prescriptions of the foundry. Native assets can be transferred by any UTXO. This way, native assets are first-class citizens in the IOTA ledger. They can be owned and transferred using the usual mechanisms by wallets and smart contracts. They also bear the metadata of their foundry, so they can be interpreted in any way in the smart contracts.

The concept of **token wrapping** is a way to interpret native assets. Let's imagine a

smart contract similar to ERC-20 on the smart contract chain. It is a common tokenization pattern – for example, that of Ethereum.

On the ISC chain, such a contract will have a foundry output owned and controlled by the ERC-20 contract, with the supply schema that bears the identity of the chain and smart contract itself. Upon request, the smart contract will mint new token(s) and will **lock** respective tokens (essentially destroying IOU rights) owned by the requestor on the smart contract. It means that L1 now has one new token with the identity of the chain and the ERC-20 contract. The smart contract will keep the tokens locked until the minted token is returned from L1. Otherwise, the newly minted native asset freely circulates among addresses on the UTXO ledger without taking into account the locked tokens in the smart contract on the specific chain.

The advanced tokenization features provide a medium for rich interoperability between smart contracts of different chains/ledgers. The **wrapping** can be given all kinds of semantics, not only coins and tokens. For example, electricity sensor readings can be tokenized into tradable assets with traceable proofs of origin.

11 Annex: Assumptions about L1 UTXO ledger

ISC comes with significant extensions of the UTXO ledger, known as the **UTXO types**[14].

It must be noted that, as a protocol, ISC, as a protocol, only makes assumptions about the structure and function of the ledger on L1. It does not make assumptions about the consensus protocol on L1.

Here we informally describe the model of the UTXO ledger on L1 which provides support for ISC chains and advanced tokenization features.

11.1 Introduction

The **ledger state** consists of *Unspent TransaXtion Outputs* (**UTXO**s): $S = \{U_1, U_2, ..., U_N\}$. Each output has an ID and is unique on the ledger. Each output has the output type and contains output data (metadata). The balances of assets and target address to where those assets are sent by the output, are examples of an output data.

The UTXO transaction is a collection of inputs, outputs and unlock data: $Tx = (IN, OUT, Unlock)$. Here:

- $IN = IN(Tx) = \{I_1, ..., I_N\}$ is a list of references to consumed (or 'spent') UTXOs. We say those outputs are consumed by the transaction.
- $OUT = OUT(Tx) = \{U_1, ..., U_M\}$ is a list of new UTXOs produced by the transaction.
- $Unlock$ is a data which unlocks the inputs, usually by some kind of cryptographic proof of authorisation to consume/spend assets on the outputs listed in $IN(Tx)$. In

[14] RFC #38: Output Types for Tokenization and Smart Contracts: https://github.com/lzpap/protocol-rfcs/blob/master/text/0038-output-types-for-tokenization-and-sc/0038-output-types-for-tokenization-and-sc.md

most common case $Unlock$ is the digital signature(s) of inputs and outputs, verifiable with addresses of inputs.

The **ledger** is a collection of UTXO transactions: $L = \{Tx_1, ..., Tx_K\}$. The ledger state $S(L)$ consists of the UTXOs which are not consumed by any transaction on the ledger, hence the term 'unspent transaction outputs'.

The **output type** defines specific **ledger constraints** which the output brings to the ledger when its containing transaction is added to the ledger. By $constraints(S, U) = true$ we denote the fact that all constraints imposed by the output U (depending on its output type) are consistent with the ledger.

By definition the **ledger state** $S(L)$ **is consistent** i.e. $consistent(S(L)) = true$ only if $constraints(S, U) = true$ for all UTXOs in the ledger state: $U \in S(L)$.

The ledger constraints are enforced by transaction validation rules. One may see output types as hardcoded validation scripts.

The ledger is an append-only structure: this means that we can only add transactions to it. Each transaction is therefore an atomic update of the ledger state. Let us say that we have ledger L and the ledger state is consistent i.e. $consistent(S(L)) = true$. By adding the transaction to the ledger we modify the ledger state and the state transition occurs: $L_{prev} \cup \{Tx\} = L_{next}$. This is only valid if the transaction Tx satisfies all the validity conditions:

- Tx is syntactically valid.
- Inputs $IN(Tx)$ references existing UTXOs on the ledger state $S(L_{prev})$, i.e. UTXOs which are not consumed by any transaction already existing on the ledger.
- $Unlock(Tx)$ is a valid unlock, i.e. signatures are valid.
- State transition preserves consistency of the ledger state: $constraints(S(L_{next})) = true$ and this fact can be determined by only using context of the transaction Tx, i.e. its inputs, outputs and unlock blocks.

The important property of the UTXO ledger is that validity of the state update, a transaction, can be determined by the transaction itself, without access to the global state of the ledger.

The above implies that each output can be consumed by at most one transaction, otherwise the two transactions would make a **double spend**. Ledger consistency rules do not allow double spends and the ledger does not have double spends.

The rules above also make UTXO ledger partially ordered structure: by the very way how it is constructed it cannot contain cycles. The UTXO ledger starts at the **genesis**, the transaction and outputs, which is the ultimate predecessor of any transaction of the UTXO ledger.

The ledger at genesis L_0 is consistent by assumption. The consistency of each state transition $L_i \rightarrow L_{i+1}$ guarantees consistency of each ledger state. We can talk about a ledger invariant $I(L) = const$ which is preserved by each transaction (transition of the

ledger state). In the genesis state the invariant is defined by the genesis transaction: $I(originLedger) = const$. Later, when updating the ledger by adding a new transaction, all conditions (constraints) of the ledger invariant remain satisfied. For example, among other things, the IOTA ledger invariant maintains the constant supply of IOTA tokens defined in genesis: $2,779,530,283,000,000$.

11.2 Scalability of the UTXO ledger

An important property of the UTXO ledger is that the structure of it is parallelizable and therefore scalable. Each transaction, the ledger state update, only consumes a fixed collection of UTXOs, i.e. it only depends on the fragment of the overall ledger state. Two transactions Tx_1 and Tx_2 can both be added to the ledger only if they do not have common inputs, i.e. $IN(Tx_1) \cap IN(Tx_2) = \emptyset$.

It is easy to check if a transaction is not a double spend. Also, independent transactions **can be added (written) to the ledger in parallel** because they won't produce conflicts (double spends). This property makes parallel ledger updates possible.

11.3 Ledger, ledger state and pruning

We distinguish two things: UTXO ledger L itself and UTXO ledger state $S(L)$. **The main purpose of the ledger is to keep the consistency of the ledger state**. The ledger itself, the history of transactions, is not critical for the operation of the ledger; however, it is necessary for auditability purposes, as a history of transactions which led to the current ledger state, starting from genesis.

Once an output is spent by a transaction it is no longer present in the ledger state and can be deleted (or 'pruned'). The same is true for transactions: transactions are atomic groups of outputs that maintain links between inputs and outputs. So, transactions are only needed for the atomicity of the ledger state update; after a transaction is added to the ledger, it is only needed for auditability purposes, i.e. to prove the ledger state transition is valid. Otherwise, transactions can be forgotten (i.e.deleted), even if not all their outputs are spent.

In conclusion, the ledger state is always available, while the whole ledger is not.

11.4 UTXO ledger accounts

An **account** is a subset of the ledger state. It is a collection of UTXOs that are **controlled** by some entity. Usually the entity controls the account via a private/public key pair. The account is identified in the ledger by its **address**. To prove its right to control the account, the entity provides a signature through the private key and the protocol rules check if the signature corresponds to the address.

In an account-based ledger such as Ethereum, an account is essentially a balance of a token and data: this is known as the account state. The **account state** can be modified (or controlled) by the owner of the address (i.e. the private key). In the account-based ledger the account is global and always has an objective state. If the owner of the account is a smart contract, it always receives the same account state as an input for its program, no matter on which node the program is executed. For the smart contract, the state is

deterministic in the context of a specific block.

In contrast, an account in the UTXO ledger consists of UTXOs (unspent outputs) which have a particular address as its target address. Parallel writing to the ledger also means parallel writing to the account state. So, in the UTXO ledger the state of the account is not objective: different nodes may see the overall collection of UTXOs in the account slightly differently (unless they put them into blocks, like in Bitcoin or Cardano). So, programs which consume the state of the same UTXO account on different nodes may have different perceptions of it and provide different inputs for the program.

The above means smart contracts which process the state of the account first have to deal with the non-deterministic nature of the UTXO account.

11.5 UTXO extensions

In the IOTA UTXO ledger the **output type** determines constraints that the output brings to the ledger. One may see the output type as a hardcoded **configurable constraint validation script**, or **state machine**, which enforces certain constraints in the ledger by taking transaction context and the UTXO as input and producing *true* or $false$ as a result of checking the constraints. The UTXO (and the transaction) is valid only if the result is *true*, i.e. does not allow a ledger state with invalid constraints.

The simplest type of UTXO model only assumes asset balances to be present in the output. The constraint of such an output is preserving token balances in inputs and outputs; for example, to preserve a constant number of tokens in the ledger.

We are extending the IOTA UTXO ledger by adding special output types. These are designed to program the UTXO ledger by adding other constraints to it in order to support L2 ledgers on top of the IOTA UTXO ledger with advanced tokenization features. Here we informally provide an incomplete list of the output types. For detailed information, we refer to RFC #38: Output Types for Tokenization and Smart Contracts.

An alias output is an output type which allows building non-forkable chains controlled by a state controller address specified in the output itself. The alias output is minted (i.e. newly created) and the protocol assigns it a random and unique ID, known as the **alias address**, for the duration of its lifetime. The constraint it brings to the ledger guarantees that there is always one – and only one – alias output with the given ID on the ledger. We can always query the alias output by its ID on the ledger.

The alias output can be unlocked by the valid signature of the state controller address. Once unlocked as an input in the transaction, one must put exactly one alias output with the same ID in the outputs of the transaction. The new alias output can modify the metadata and the state controller address. This way, by consuming alias output, the state controller (i.e. the private key behind the address) can build non-forkable chains.

The alias output implements a feature of address aliasing on the ledger: namely, the ability to change controlling entities (i.e. private keys) of the account, without changing the address, i.e. transparently to the users of the address.

ISC uses alias outputs for state anchoring, which means committing to the globally unique state of the chain on the UTXO ledger state. By producing the next output in the chain we atomically record the next state of the chain on the UTXO ledger (L1). The address aliasing also makes the identity of the chain decoupled from the identity of the state controller.

Extended value transfer output (also known as **extended output**) extends the simple value transfer output type with configurable validation options. It allows transfer of IOTA tokens and native assets to a target address, which can be any type of address, including the alias address.

Extended value transfer output can be consumed by either providing a signature of the address, or, if the target address is an alias address, unlocking the corresponding alias output in the same transaction. This way, the state controller of the alias output also controls all extended outputs with the target address of the alias output.

ISC uses extended value transfer output for on-ledger requests between chains and addresses. The metadata of the output also bears parameters of the request. The output is consumed in the same transaction as the state transition by the alias output. The semantics of consuming the output is equivalent to "processing the on-ledger request" to the chain.

The extended value transfer output configuration options also include timelock, expiry deadline for its consumption, and more.

Foundry output is a special output type which extends the ledger with new types of tokens, the so-called *native assets*. It enables the minting of new supplies of tokens and the controlling of them through inflating (creating from thin air) or deflating (destroying the ones in the possession of the foundry). One foundry output corresponds to the supply of a token on the ledger and the ledger enforces consistency of the token accounts: only the foundry can create or destroy a specific token. The foundry can be seen as a central bank with its own "monetary policy". The monetary policy in this case is an immutable property of the foundry set upon its creation. It is enforced by the protocol and foundry output type validation rules. The examples of specific "monetary policy" can be "no inflation, only deflation", "fixed supply", "no more than 10% annual inflation" or similar. The foundry output itself is controlled by the alias address that created it; in other words, control of the supply of a specific token ultimately goes to the controlling private key of the alias output.

NFT output is similar to the alias output, so its constraints guarantee that there is always one NFT output with a specific ID on the ledger. NFT stands for *Non-Fungible Token*. Just like any UTXO, the ownership of the NFT output belongs to an address, so it can be owned and transferred between accounts on the L1. Another essential property of the NFT output is that it contains an immutable **proof of origin**, enforced by the ledger constraints: whoever creates (or 'mints') the NFT output, their public key becomes an immutable part of the NFT output for the duration of its lifetime.