Tapyrus Technical Overview

Shigeyuki Azuchi
chaintope, Fukuoka, Japan

Draft

Abstract

Since the introduction of Bitcoin in 2009, a number of distributed ledger technologies have emerged. These allow secure payments based on cryptographic assumptions instead of a central administrator as in traditional systems. In addition, ledger technologies such as Ethereum have emerged which allow arbitrary logic to be executed on a VM while the results being shared. As a result, it is attracting attention not only as a simple payment system but also as a platform for cross-enterprise business use in place of traditional IT systems. In particular, as for blockchain usage for enterprises, there have been a number of PoCs conducted using permission-based blockchain such as Hyperledger Fabric, which limits the number of participating nodes. There are several challenges of using blockchain technologies, which are fully opened such as Bitcoin and Ethereum, for enterprise applications. For example, lack of finality, high volatility of the base currency, and lack of governance regarding chain operations and additional features. On the other hand, as for a permission-based blockchain in which the population of nodes participating in the network and verification is limited, the authenticity of the ledger data is proved also to a limited number of participants. Users can not directly be part of the consensus mechanism of the blockchain, and the system tends to be centralized via APIs, etc., which can be replaced by typical IT systems such as RDB.
Our product, Tapyrus, is a hybrid blockchain which solves the challenges when enterprises adopt blockchain while maintaining the openness of blockchain. It solves these issues by separating the federation layer from the ledger network. Federation layer generates blocks (approves transactions) and maintains, also operates the chain. On the other hand, ledger network provides open access to the ledger data. This paper describes the structure of Tapyrus, the features provided, and the Layer2 protocol which can be configured on top of Tapyrus.

Keywords: Blockchain; DLT.

1 Introduction

Since the introduction of Bitcoin [1] in 2009, a number of distributed ledger technologies based have emerged. In traditional payment systems, transactions are approved by a centralized server operated by an administrator, while Bitcoin does not have such centralized

server and transactions are approved in an open network where anyone can participate. Before Bitcoin appeared, it was considered that it was difficult to approve unique transactions in such a decentralized network, however, Bitcoin solved this problem through a computational competition called Proof of Work (PoW) and rewards given as the result. Since the invention of Bitcoin, a number of blockchain products have emerged, which not only work as a simple payment application, but also provide token protocols that allow the issuance and transfer of arbitrary tokens, and smart contracts that allow arbitrary logic to be executed on VM. It was considered that such blockchain technology could be widely used not only as cryptocurrency, but also as enterprise application, and the potential of blockchain was explored. The following issues exist when using these blockchains in the enterprise domain.

Lack of finality There is no finality in PoW consensus mechanism. Even if a transaction is stored in a block, it can be replaced by another block later due to chain reorganization. Therefore, even a transaction stored in a block doesn't simply guarantee that the transaction is confirmed. However, as subsequent blocks are connected to the block in which the transaction is stored, the probability of a reorganization occurring becomes small. This is why it is called probabilistic finality. Moreover, if a chain adopts the PoW algorithm but with low hash power and the hash power is easily transferable, there is also a possibility of an attack that could result in a large scale reorganization of the chain [2].

High volatility In a public blockchain, each transaction requires a fee for the transaction to be executed. If there is no fee setting for processing a transaction in an open network, the cost of issuing a transaction will be effectively zero, however, it actually costs the participating nodes to execute the transaction. As a result, Denial-of-Service (DoS) attacks are possible by injecting a large number of transactions into the network, which result in bringing down the blockchain network. For this reason, public blockchain transactions require a fee setting, which should be paid in the base currency of the chain. However, cryptocurrencies are currently subject to high volatility, making it difficult to estimate the actual cost when using them in enterprise applications. In addition, external factors such as the emergence of popular smart contracts will cause the number of transactions to increase in the network, causing transaction fees rise remarkably to store them into blocks.

Governance In blockchains with no governing body, it is difficult to build consensus on how to implement feature enhancements which may affect the consensus mechanism or how to tackle a problem when fatal flaws are found. In Bitcoin, a conflict between miners and developers occurred related to the soft fork for adding features in 2017, which result in a long delay in activating the soft fork, and subsequently also had to be cautious to add features (the introduction of Taproot) which took four years for the introduction. Furthermore, the features proposed in the meantime are also not yet ready for deployment. In 2016, Ethereum underwent a hard fork to tackle The DAO incident, in which a large amount of funds were stolen in an attack using a bug in smart contracts. While there were also other problems caused by the smart contract bug, it was considered as a problem to rescue this single case, and as a result, the opponents decided to segregate the chain. A new chain was born which is known as Ethereum Classic. In such a chain without an administrator, governance of the chain's operations becomes an issue.

Then came the Hyperledger Fabric [3], a permissioned blockchain with limited participants which can avoid these challenges for enterprise applications. It used Practical Byzantine Fault Tolerance (PBFT) as a consensus algorithm. This is an algorithm that requires the population of participants to be determined in order to reach a consensus, and this is different from algorithms such as PoW where anyone is able to participate without any restriction. In a permission-based blockchain, the number of nodes that can participate is limited, and ordinary users basically use the service through APIs provided by the node operator. Whether a user is an appropriate user or has the necessary privileges is implemented arbitrarily at the API layer, therefore, countermeasures against Denial-of-Service attacks are taken here, and transaction fees are not required at the blockchain layer. On the other hand, the scope of proving the authenticity of the ledger data is also limited to the participating nodes. Since API-based access is not directly related to whether the back-end is a blockchain or an RDB from the user's point of view, it can often be substituted by conventional IT systems, and this tends to make it possible to use the knowledge of conventional IT systems and typically able to reduce cost.

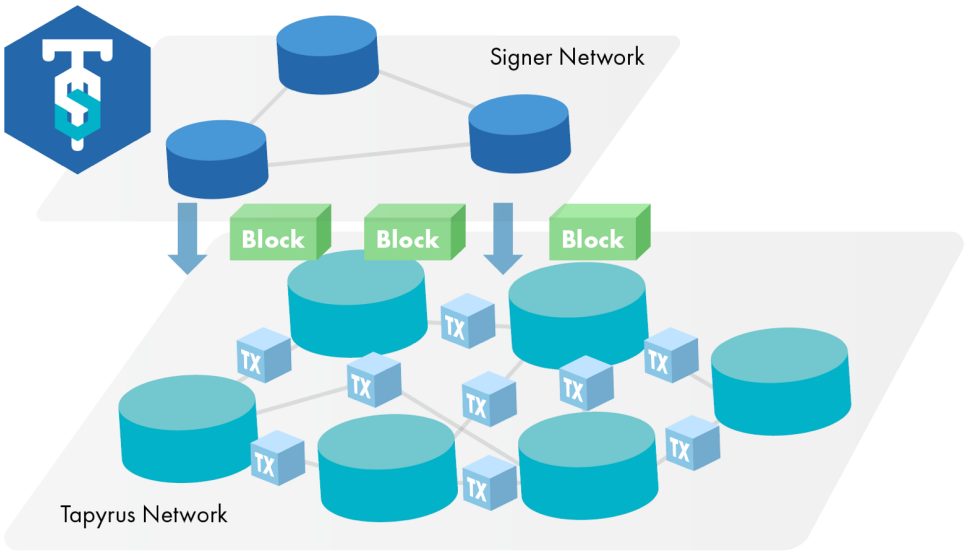

Figure 1: Tapyrus network construction

1.1 Our contributions

We believe that an important aspect of blockchain is its openness to access to the ledger. Our proposed hybrid blockchain, Tapyrus, separates the blockchain layer into a signer network layer, which generates blocks through federation which maintains and manages the chain, and a ledger network layer that anyone can join as a node. In this way, it maintains the openness of the blockchain and solves the challenges which occurs for enterprises.

2 Structure of Tapyrus

The Tapyrus blockchain consists of independent networks for each arbitrary business use case (e.g., electricity certificate trading, CO2 trading, local currency, corporate coins, etc.). As shown in Figure 1, each network consists of the Signer Network, which is operated by a federation consisted mainly by stakeholders of the business use case, and the Tapyrus Network, which is an open ledger network.

Finality In a PoW such as Bitcoin, anyone can become a miner, but in Tapyrus, the number of signers who can create blocks is limited when the chain is set up. The selected signers which are going to maintain and operate the chain has to create a digital signature for the new block and attach it to the block, then a valid block will be created on the Tapyrus Network. This means that the transaction is given finality at the time the block is created, rather than probabilistic finality as in PoW.

2.1 Block structure

Tapyrus is based on the Bitcoin blockchain with consensus algorithm changed and additional features added. One of the major changes is the replacement of the block creation rules from PoW to digital signatures by multiple signers. As a result, the Tapyrus block header consists of Table 1 elements. The *proof* field in this table is the place in which the digital signature (Schnorr signature) created by the Signers in cooperation will be included.

Field	Type	Size	Description
features	int32	4	Feature flag. Currently, 1.
hasPrevBlock	char[32]	32	Previous block hash.
hashMerkleRoot	char[32]	32	Root hash of the Markle tree consisting of the hashes of all transactions in the block.
hashImMerkleRoot	char[32]	32	Root hash of the Markle tree consisting of the TXIDs of all transactions in the block.
time	uint32	4	Block time(UNIX timestamp).
xfieldType	uint8	1	Type of xfield(see below).
xfield	N/A	variable	extension field.
proof	char[65]	65	Block signature(Schnorr signature data and data length as prefix).

Table 1: Tapyrus BlockHeader structure

The currently defined *xfieldType*is as shown in Table 2.

2.2 Schnorr signature scheme

The Scuhorr signature scheme used for signing blocks and for signatures used within transactions is as follows:

Create signature The elliptic curve used in Tapyrus is secp256k1, the same as Bitcoin, and let $\mathbb{G}$ is generator of the elliptic curve of order n and let H is the cryptographic hash function.

Creating a signature takes the following items as input:

Value	Name	Type	Size	Description
0x00	None	N/A	1	None
0x01	Aggregated public key	char[33]	34	Aggregated public key used to verify Tapyrus block

Table 2: Tapyrus BlockHeader structure

- 32 bytes array as private key sk
- 32 bytes array as message digest m

Using the private key sk, a signature for message m is created by the following procedure:

1. Let $d' = int(sk)$.
2. If $d = 0$ or $d => n$, fails. Where n is order of elliptic curve.
3. Let $P = d'\mathbb{G}$.
4. Generate random nonce k', and let $R = k'\mathbb{G}$.
5. If $jacobi(R.y) = 1$, let $k = k'$, otherwise let $k = n - k'$.
6. Let r be the x-coordinate of R.
7. Let $e = int(H(bytes(r)||bytes(P)||m)) \mod n$.
8. Calculate $s = (k + ed) \mod n$.
9. Signature data is (r, s).

Verify signature The signature verification takes the following items as input:

- Public key $P = sk\mathbb{G}$ corresponding to private key private key sk
- 32 bytes array as message digest m
- Signature data (r, s)

Using these data, the following procedure is used for verification:

1. If public key P does not exist on curve or refer to infinity point, verification fails.
2. If r is larger than the field size of the elliptic curve, verification fails.
3. If s is larger than order of elliptic curve, verification fails.
4. Computes $e = int(H(bytes(r)||bytes(P)||m)) \mod n$.
5. Conputes $R' = s\mathbb{G} - e\mathbb{P}$.
6. If x-coordinate of R' equals to r, verification success.

2.3 Signer Network

The Signer Network is a network for maintaining and operating the blockchain, operated by members of a federation predetermined at the time the network is set up. Specifically, it collects and verifies new transactions in the Tapyrus Network, creating a new block containing those valid transactions, and broadcasts it to the Tapyrus Network.

2.3.1 Block verification

Whether or not the block created by the Signer Network is a valid block can be confirmed by verifying that the *proof* of the block is a valid signature. The data used for this verification is as follows:

- Signature data (r, s) set in *proof* of block.
- Let the double-SHA256 hash value of the data excluding *proof* from the block header of the table 1 be the message m to be signed.
- Aggregated public key.

The aggregated public key uses the value stored in *xfield* in the block header of the genesis block. This aggregated public key is the sum of the public keys of all Signers. If a new aggregated public key is set in a subsequent block using *xfieldType = 0x01*, then the subsequent block will use that new aggregated public key for signature verification. Since the aggregated public key is the data which is available to all participants in the Tapyrus Network, it is possible for all participants in the network to independently verify the correctness of a block.

2.3.2 Verifiable threshold signature scheme

In the Tapyrus Signer Network, multiple Signers cooperate to create a signature for a block. Because of the simplified representation in the signature scheme described above, an aggregate private key sk corresponding to an aggregate public key P has been introduced. However, the aggregate secret key itself is the data obtained by adding the secret keys of all Signers, and in order to calculate this value, all Signers need to disclose their secret keys, which is undesirable. Therefore, Tapyrus Signer Network creates aggregated signature based on DR. Stinson and R. Strobl "Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates" [6]. This creates a valid Schnorr signature for the aggregated public key without revealing the private key of each Signer (i.e., without creating an aggregated private key sk). This scheme uses a combination of the Feldman scheme Verifiable Secret Sharing (VSS)[7] and Pedersen's verifiable secret sharing method with Schnorr signatures. This provides a threshold signature that can be used to complete a valid Schnorr signature when a threshold t partial signatures are collected from n Signers. Since it is not necessary to collect partial signatures of all Signers, but only t, it is robust against partial Signer failures within the threshold.

The threshold value can be set arbitrarily, but it is desirable to set a value that exceeds the majority or satisfies Byzantine tolerance for the population. Since the Schnorr signature generated by this signature scheme is a single Schnorr signature from the point of view of the network participants, the signature verification cost for each participant remains constant even if the population of Signers increases or decreases. On the other hand, since signature generation requires the exchange of partial signatures generated by each Signer, the cost of this message exchange increases proportionally as the population of Signers increases.

2.3.3 Signer Network flow

Each signer node is identified by the public key that each one holds. An index is assigned to each node in the lexical order of this public key, and the round master (see 2.3.3) is determined based on this index.

The creation of a new block is coordinated by the master of that round, elected from each Signer, and the other members. A valid *proof* is generated for the block if the Signer's

agreement that meets the threshold is reached in that round, and the created block is broadcasted to the Tapyrus Network. After that, the next round of block generation begins, and block creation continues. The following are the specific messages and algorithms exchanged between Signer nodes.

Message type The communication between each node is done using messages broadcasted to the Redis pub/sub. The message types currently defined are as shown in Table 3.

Message	Payload	Description
candidateblock	Block	Round Master sends candidate block.
blockvss	BlockVSS	Send vss of random secret
blockparticipants	Vec$< PublicKey >$	Notification of signature protocol by round master.
blocksig	LocalSig	Send partial signature.
completedblock	Block	Round master send completed block.

Table 3: Message types communicated between Signers

Block creation round process The period of time it takes for Signers to exchange messages and create a single block is called a round. For each round, a Round Master is elected from among the Signers. The round master is elected by round robin. The communication flow between the Round Master and each Signer in the block creation round is shown in Figure 2.

This signing process is implemented in Rust in Tapyrus Signer[8].

2.4 Tapyurs Network

The Tapyrus Network is a blockchain network that anyone can join, providing all on-chain data for the blockchain from genesis blocks to the latest blocks, and supporting the propagation of new transactions. Tapyrus' full node Tapyrus Core[9] is implemented by forking Bitcoin Core[10], the reference implementation of Bitcoin.

The Bitcoin-based blockchain was chosen for the following reasons:

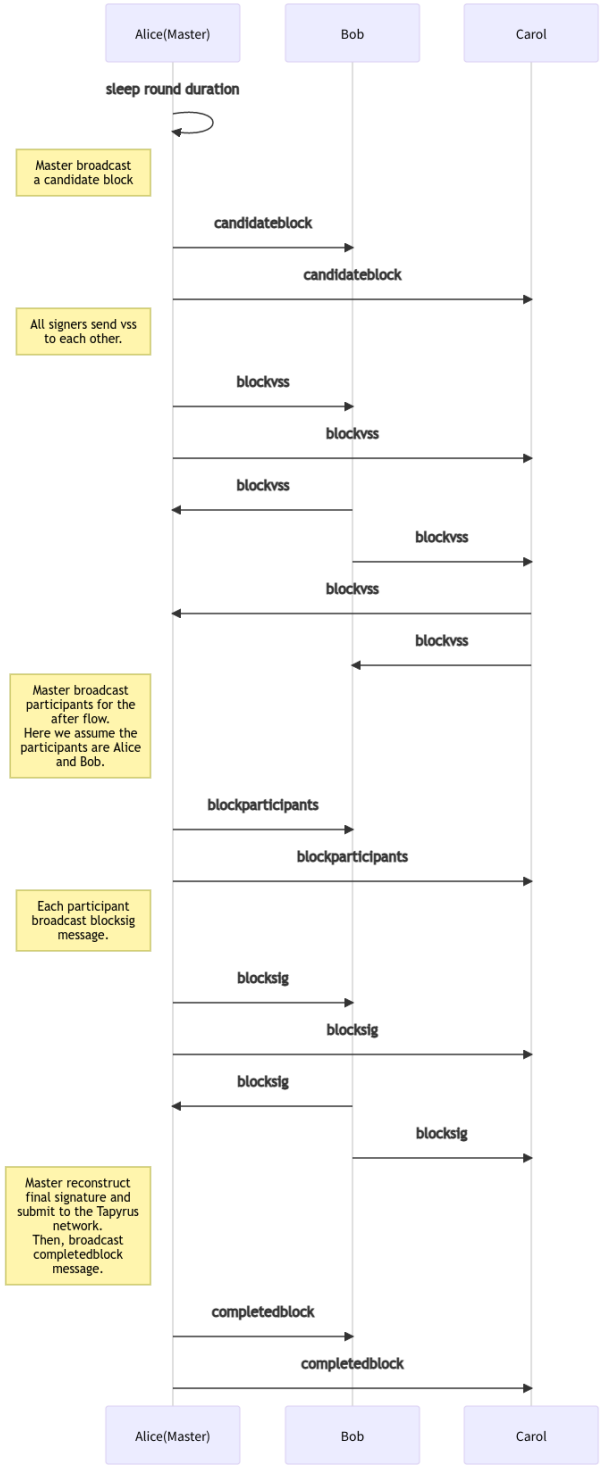

Figure 2: Processing the signature generation round.

- Constraints such as block size limits and the maximum number of high-cost opcodes used in a transaction within a block make the speed of data growth and the cost of script verification predictable. This is an important factor in the stable operation of the system.

- The P2P code has been in operation since 2009 without stopping while being exposed to various attacks. Although it is not a consensus specification, the code implementation, which exchanges P2P messages to build a blockchain, implements a variety of measures against attacks.

2.4.1 Tapyrus Core

Tapyrus is basically based on Bitcoin's transaction [4] and Script [5] systems with a few improvements.

Changes in consensus algorithm Changed the consensus algorithm for PoW and best chain rules to a consensus algorithm based on digital signatures generated by the Signer Network. While Bitcoin's block generation interval is adjusted to be about 10 minutes by the difficulty adjustment algorithm, Tapyrus' block generation interval can be adjusted arbitrarily by Signer.

Fix TXID malleability A transaction has an identifier, called TXID, which uniquely identifies the transaction in the blockchain. This identifier is not a value that is explicitly set to the transaction, but is a value calculated from a double-SHA256 hash of the transaction data and expressed in little-endian. Therefore, if any part of the transaction data is changed, the TXID will also change. Since transaction data is usually protected by digital signatures, transaction data cannot be changed, but only the digital signature data can be changed since it cannot be signed. Changing the signature data would normally result in an invalid digital signature, but by exploiting the malleability[1] of ECDSA itself and changing the way the opcode is specified when setting the signature, it was possible to change the TXID while keeping the digital signature valid.

[1] The ECDSA signature value (r, s) is also valid signature data as $(r, -s \mod n)$

In Bitcoin, this has led to a problem where the TXID of a transaction broadcasted on the chain can be unintentionally changed by a malicious user. For simple on-chain payments, this is not a big problem, but for Layer 2 protocols such as Lightning Network[11], it is necessary to build transactions with parent-child relationships and pre-sign each one. In order to guarantee the parent-child relationship of the transactions, the malleability of the TXID needs to be resolved. Bitcoin addressed this with Segregated Witness, which was activated in 2017. Tapyrus fixes this vulnerability by separating the transaction hash from the TXID and not including the signature data in the TXID calculation.

Schnorr signature support In addition to block verification, Tapryus supports Schnorr signatures in addition to ECDSA for the signature verification algorithm used in the Script system for transactions. Specifically, ECDSA and Schnorr signatures are evaluated in the following four opcodes:

- OP_CHECKSIG
- OP_CHECKSIGVERIFY
- OP_CHECKDATASIG
- OP_CHECKDATASIGVERIFY

Which algorithm is used will be determined by the size of the signature data provided. The availability of Schnorr signatures in Script has the following advantages: a) The signature data itself can be smaller, b) Multi-sig signature data, which requires multiple users' signatures to use a coin, can be aggregated into a single signature data, c) technologies such as Scriptless Script[12], which allows for the implementation of conditional contracts in the digital signature itself without the use of Script, can be more easily implemented. In addition, the Schnorr signature has a formal security assessment and there is no signature malleability like ECDSA.

Oracle support Smart contracts running on the blockchain are essentially run in a sandbox environment, so it is not possible to run a contract that relies on external data(real-world data). This is because there is no way to verify whether the data is correct or not in

the blockchain consensus. In contrast, by setting up a trust point (called an oracle), which is the provider of the data, it is possible to realize a contract that relies on external data. In Tapyrus, the following two opcodes have been added so that the oracle can be used.

- OP_CHECKDATASIG
- OP_CHECKDATASIGVERIFY

These opcodes verify signature for arbitrary messages. These opcodes take the following three elements as input.

1. Public key
2. The arbitrary messages
3. Signature(ECDSA or Schnorr)

It then calculates the SHA-256 hash value of the message and uses it as the message digest to verify that the signature is a valid digital signature for the public key. When executing a contract using Oracle, the public key is Oracle's public key, the message is the data used for the conditions of the contract published by Oracle, and the signature is the signature generated by Oracle. The reference implementation of both opcodes is shown in Listing 1.

Listing 1: Reference implementation of OP_CHECKDATASIG and OP_CHECKDATASIGVERIFY

```
case OP_CHECKDATASIG:
case OP_CHECKDATASIGVERIFY: {
    // (sig message pubkey -- bool)
    if (stack.size() < 3) {
        return set_error(
            serror, SCRIPT_ERR_INVALID_STACK_OPERATION);
    }

    valtype &vchSig = stacktop(-3);
    valtype &vchMessage = stacktop(-2);
    valtype &vchPubKey = stacktop(-1);

    //check signature encoding without hashtype byte
    if ( (vchSig.size() != CPubKey::COMPACT_SIGNATURE_SIZE - 1
         && !CheckECDSASignatureEncoding(vchSig, serror, true))
        || !CheckPubKeyEncoding(vchPubKey, flags, sigversion, serror)) {
```

```
17          // serror is set
18          return false;
19      }
20
21      bool fSuccess = false;
22      if (vchSig.size()) {
23          valtype vchHash(32);
24          //Hash message
25          CSHA256().Write(vchMessage.data(), vchMessage.size())
26              .Finalize(vchHash.data());
27          //no hashtype in signature. call VerifySignature not CheckSig
28          fSuccess = checker.VerifySignature(vchSig, CPubKey(vchPubKey),
                uint256(vchHash));
29      }
30
31      if (!fSuccess && (flags & SCRIPT_VERIFY_NULLFAIL) && vchSig.size()) {
32          return set_error(serror, SCRIPT_ERR_SIG_NULLFAIL);
33      }
34
35      popstack(stack);
36      popstack(stack);
37      popstack(stack);
38      stack.push_back(fSuccess ? vchTrue : vchFalse);
39      if (opcode == OP_CHECKDATASIGVERIFY) {
40          if (fSuccess) {
41              popstack(stack);
42          } else {
43              return set_error(serror, SCRIPT_ERR_CHECKDATASIGVERIFY);
44          }
45      }
46  } break;
```

The difference between OP_CHECKDATASIG and OP_CHECKDATASIGVERIFY is the same as the difference between OP_CHECKSIG and OP_CHECKSIGVERIFY, which is whether to push the result to the stack or to evaluate the Script as a failure as soon as it fails.

Token protocol The issuance and transfer of arbitrary tokens is a well-known feature in blockchain use cases. In Bitcoin, a number of overlay protocols (such as, Open Assets Protocol[13] and the Counterparty Protocol[14], etc) were developed early on using the OP_RETUN opcode, which can record 80 bytes of arbitrary data. The ERC20 Token has a particularly large share in Ethereum.

While most protocols are overlay Layer 2 protocols, Tapyrus supports the issuance and transfer of arbitrary tokens in Layer1 in addition to the native token (TPC/tapyrus).

Overlay protocols are Layer 2 protocols that use the ability to record arbitrary data into the blockchain which enables value transfer separate from the native token. They are not verified for correctness as a consensus of Layer1, and the tokens are recognized by wallets that interpret the protocol independently, called Client Side Validation. This means that if the data is passed to a wallet that does not interpret them, they will be interpreted as Layer1 native token, not tokens. Also, to verify the correctness of these tokens, we need all the transfer transactions from the originating transaction where the token was issued, and we have to keep the already used TXO (Transaction Output). This means that the pruning feature to delete the used TXO data of a node is not available, which puts pressure on the disk size of a full node. Also, the need for historical transactions makes it unsuitable for verification on lightweight devices such as smartphones and IoT devices. Tapyrus supports the token protocol as a Layer1 feature to facilitate pruning of used TXOs and identification of tokens on lightweight devices. There are three types of tokens that can be issued on Tapyrus:

- Re-issuable Token
- Non re-issuable Token
- NFT (Non-Fungible Token)

Token specification Tapyrus Script has an additional opcode, OP_COLOR opcode, to identify the token in Script, and its specification is shown below.

When the OP_COLOR opcode appears in a Script, the top element of the stack is interpreted as the COLOR identifier. If there is no element left on the stack, or if the COLOR identifier does not follow the rules described below, the script will fail. If the COLOR identifier conforms to the rules, the coins in its UTXO represent the amount of coins in that COLOR identifier. A Script that does not contain OP/_COLOR refers to the amount of the native token TPC (tapyrus).

In a transaction, the total amount of tokens in the input color identifier and the total amount of tokens in the output color identifier basically match (except for token burning). In other words, the balance of each token in a transaction is maintained. At this point, it is assumed that the Signer will not receive any tokens as a fee for block generation.

If OP_COLOR is used in Script, the following restrictions apply:

- Only one OP_COLOR may be included in a Script.
- OP_COLOR cannot be placed in a branch of a control opcode such as OP_IF.
- The scriptPubkey of the custom token for each output will always contain the OP_COLOR. For this reason, it is not possible to include OP_COLOR in a P2SH redeem script. If such a script is configured, the script interpreter's stack will contain multiple OP_COLORs, which will always result in an error and lost funds when using P2SH.

As a result, we support the following types of scriptPubkey, which are colorized versions of the existing P2PKH and P2SH.

- CP2PKH(Colored P2PKH) :
  $< COLOR\ identifier > OP\_COLOR\ OP\_DUP\ OP\_HASH160 < H(pubkey) > OP\_EQUALVERIFY\ OP\_CHECKSIG$
- CP2SH(Colored P2SH) :
  $< COLOR\, identifier > OP\_COLOR\, OP\_HASH160 < H(redeemscript) > OP\_EQUAL$

As described above, the token UTXO always has the COLOR identifier of the token, which makes it possible to identify the token by the UTXO alone. This ensures that unused tokens are retained even when the blockchain is pruned of used data, and lightweight nodes can recognize the tokens only by the UTXOs they receive.

The COLOR identifier consists of a 1-byte type and a 32-byte payload. The type and payload currently supported by the COLOR identifier are shown in Table 4. The token issuing transaction will verify that the COLOR identifier specified by OP_COLOR in the output satisfies this rule for the data in the input. In addition, for type 0xC3, it shall additionally verify that the issue amount is 1.

The above rules ensure that Non re-issuable Token and NFT that cannot be reissued with the same COLOR identifier, since their COLOR identifier is generated from the Out-Point that the input of the issuing transaction refers to.

Type	Name	Payload
0xC1	Re-issuable Token	32 bytes of data that is the SHA256 value of the scriptPubkey in the issue input.
0xC2	Non re-issuable Token	32 bytes of data that is the SHA256 value of the OutPoints at issue input.
0xC3	NFT	32 bytes of data that is the SHA256 value of the OutPoints at issue input.

Table 4: COLOR identifier type

Token issuance transaction When a token is newly issued, a UTXO for issuing the token shall be set to the input and a COLOR identifier shall be derived from the UTXO based on the above rules. Create a transaction with scriptPubkey(CP2PKH, CP2SH, etc) using the COLOR identifier and OP_COLOR opcode in the transaction output.

The output of issuing a new token can be set to an arbitrary amount of tokens as value. The TPC of the UTXO specified in the input will also create and collect another non-token output.

Token transfer transaction To transfer a token, create a transaction with a Token UTXO as the input, and add the output with the same COLOR identifier as the input token and the OP_COLOR opcode to the destination address.

The number of inputs, the number of outputs, and the type of tokens can be set, and it does not matter if the total amount of each type of token is maintained in the inputs and outputs.

Burn token To burn a token, create a transaction with a UTXO with the token to be burned and a UTXO with a TPC for the fee as inputs, and add an output that receives TPC change. Since the value of a UTXO for a token is entirely the amount of the token, in order to set the fees, it is necessary to set the UTXO of TPC. It is also possible to combine the above three token processes into a single transaction.

Each combination and valid/invalid pattern described in the following:
https://docs.google.com/spreadsheets/d/1hYEe5YVz5NiMzBD2cTYLWdOEPVUkTmVIRp8ytypdr2g/

3 Extension Protocol

Tapyrus is a Layer1 blockchain implementation, the same as Bitcoin, Ethereum, and Fabric. While the Layer1 protocol is sufficient for simple token transfer and exchange, but it is difficult to implement the following applications: a) high-throughput payment applications based on a P2P-based distributed network, b) applications that require transaction confidentiality, etc. Therefore, it is necessary to extend the protocol design of the upper layers according to the application. In Tapyrus, we have designed extended protocols as necessary to implement our previous use cases and products such as Paradium, and this section introduces some of these protocols.

3.1 Tracking Protocol

One of the use cases for blockchain is supply chain traceability, which tracks the movement history of objects. By recording the movement of objects in a ledger that can be freely accessed by anyone, the record of movement across companies and countries can be freely verified by anyone at any time, making it a suitable use case for open blockchain. On the other hand, the size of data that can be recorded in a blockchain is limited, even if it were possible to increase the size of data, increasing the data in a distributed network would result in a trade-off with higher node operating costs for participants. Therefore, when a large amount of movement of objects is recorded in a blockchain, how to make it compact while the data trace is provable is an important issue in blockchain throughput and disk space. In this section, we introduce an extension protocol that addresses these issues.

3.1.1 Tracking transaction

In Tapyrus, a UTXO-type blockchain, the input of a transaction consumes an existing valid UTXO, and the output of a transaction generates a new UTXO. In use cases such as traceability, it is possible to represent the input as the source of the movement of objects and the output as the destination.

Tracking payload In addition to the destination output, a tracking transaction has one extra output with a tracking payload that records information about the moving objects.

The presence of this output indicates that this transaction is a tracking transaction. The output of the tracking payload will be in the following format using OP_RETURN.

$$OP\_RETURN < Tracking\ Payload >$$

The tracking payload consists of the data in Table 5.

Field	Size	Content
Marker	2 bytes	Always 0x5450 which indicates hex value representing tracking protocol(TP)
Version	1 byte	Currently 0x01.
Payload size	CompactSize [16]	Payload size
Payload	variable	Accumulator value created with the unique identifier of the object to be moved (see 3.1.4)

Table 5: Tracking payload data

3.1.2 Trace rules

In the tracking protocol, the source and destination are determined by the following rules.

Source is represented by transaction inputs and is determined by the following rules:

- To start a new trace

  The first input in the transaction input is used as the address of the source.

- In the middle of a move

  All UTXOs to which the tracking payload in the UTXO referred to by the input is applied shall be the source address of the move.

Destination is defined as the output that is set next to the tracking payload output. It is possible to define multiple destinations for a single tracking transaction. This means that multiple tracking payloads can be defined for a transaction. The destination of each payload shall be the UTXO after it.

3.1.3 Wallet support

A wallet that supports this tracking protocol should provide the following features.

- Parse tracking transaction:
  When a transaction containing a tracking payload is received, the transaction should be parsed as a tracking transaction.
- Tracking UTXO management:
  UTXOs that have been assigned tracking payloads need to be managed as UTXOs for tracking. These UTXOs are consumed as inputs when objects are moved.

3.1.4 Data compression using an accumulator

The tracking payload does not simply record the identifiers of moving objects, but also the accumulator values that contain the identifiers. If unique identifiers are simply registered, it would increase the payload data size linearly with the number of identifiers. An accumulator is a data algorithm that can add or delete any number of objects and query whether any object is included in that or not, but it cannot retrieve the list of items added to the accumulator.

There are a variety of such algorithms, including those using binary trees, RSA cryptography, polynomial commitment, etc. The Tapyrus tracking protocol uses the RSA accumulator. This is the result of considering the following characteristics of the RSA accumulator.

- No matter how many objects you add to the accumulator, the value of the accumulator remains constant size.
- There is no need to consider the order of objects when adding or deleting data to or from the accumulator.

3.1.5 RSA Accumulator Algorithm

The RSA accumulator is initialized by the following steps.

1. Select two huge prime numbers p and q at uniformly random.

2. Compute $N = p * q$. The bit length of N should be a safe enough value. The current recommendation is 3072 bits.

3. Construct a group with N as the modulo, and select the generator g from it.

4. Set $A_0 = g$ and initialize the accumulator.

For the initialized accumulator A_0, an item with ID=x can be added by the following steps.

1. Computes hash value $H_p(x)$, where H_p is a hash function that outputs a prime number.

2. Computes $A_1 = A_0^{H_p(x)} \mod N$ using the value calculated in Step 1 for the accumulator A_0.

3. A_1 is the updated accumulator value with item x added.

Using the above steps, various items can be added to the accumulator. Since updating the accumulator is a calculation of the modular exponentiation modulo N, no matter how many items are added, the accumulator value will never exceed N. This is the reason why the RSA accumulator is a constant size.

Whether or not an item is included in an accumulator can be calculated from the inclusion proof and the accumulator value. In the above example, the inclusion proof that x is included in the accumulator A_1 is A_0. Given an inclusion proof A_0 and an item x, the verifier can verifies whether $A_1 = A_0^{H_p(x)} \mod N$ holds for the accumulator. If it succeeds, it is proved that x is contained in the accumulator A_1. The inclusion proof can also be calculated from the inner product of the hash values of all items added to the accumulator except the item to be proved.

In addition to the basic principles of the RSA accumulator described above, it is possible to use non-inclusion proofs, which prove that an item is not included in the accumulator and batch-processing [17] introduced by Dan Boneh and Benedikt B ¨ unz of Stanford University, Ben Fisch et al.

Apply to tracking protocol In the tracking protocol, the objects to be moved in a single movement as described above are set to a single accumulator and recorded in a tracking transaction. The data held on the blockchain is compressed to a constant size accumulator value, and the inclusion proof is managed off-chain by each party in a distributed manner. This allows the record of the movement to be committed on the blockchain while the necessary data is distributed to each party, thus distributing the disk load of the distributed network to the user's nodes.

3.2 Cross-chain Swap

WIP：see attachment Tapyrus Technical Guide.

3.3 Credential Protocol

WIP：see attachment Tapyrus Technical Guide.

4 Related products

In addition to the Singer Network and the Tapyrus Network, we will provide some of the software needed to configure blockchain applications.

Tapyrus Explorer https://github.com/chaintope/tapyrus-explorer/
The block explorer for Tapyrus Network.

Esplora Tapyrus https://github.com/chaintope/esplora-tapyrus
An index server for fast retrieval of transactions and UTXOs related to them, using addresses, tokens, etc. as key. The blockchain itself is not suitable for retrieval of recorded data, and in order to retrieve data at high speed, it is necessary to create an index to retrieve them, but there is a trade-off between index creation and its data space and retrieval speed. Esplora Tapyrus is an indexing server, which is also the backend of Tapyrus Explorer.

tapyrusjs-lib https://github.com/chaintope/tapyrusjs-lib
JavaScript library supporting the Tapyrus protocol.

tapyrusrb https://github.com/chaintope/tapyrusrb
Ruby library supporting the Tapyrus protocol

rust-tapyrus https://github.com/chaintope/rust-tapyrus
Rust library supporting the Tapyrus protocol

tapyrusjs-wallet https://github.com/chaintope/tapyrusjs-wallet
A Tapyrus wallet implementation written in JavaScript for a mobile application.

[WIP]Glueby https://github.com/chaintope/glueby
In developing blockchain-based applications, there are issues such as key management, state (UTXO) handling, privacy level setting, and scaling, and we need to be properly addressed, and higher-level protocols such as Layer 2 and 3 need to be designed as necessary. In many cases, these designs require in-depth knowledge of blockchain architecture and cryptography, which is a challenge for rapid application development. Glueby is a Ruby library that provides a use-case-level API for blockchain application developers to quickly build blockchain applications using Tapyrus with minimum blockchain knowledge.

[WIP]Tapyrus API https://github.com/chaintope/tapyrus-api (Private Access only)
A REST API that enables Gluby to be used on a wide range of platforms without depending on any language, allowing users to build blockchain applications by simply making API calls without being aware of the blockchain.

rsa-accumulator https://github.com/chaintope/rsa-accumulatorrb
Ruby library implementation of RSA accumulator [17].

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” http://bitcoin.org/bitcoin.pdf, 2008.

[2] “モナコインのブロックチェーン、攻撃受け「巻き戻し」　国内取引所も警戒” https://www.itmedia.co.jp/news/articles/1805/18/news071.html, 2018

[3] Hyperledger Fabric, https://www.hyperledger.org/use/fabric

[4] Bitcoin Transaction, https://en.bitcoin.it/wiki/Transaction

[5] Bitcoin Script, https://en.bitcoin.it/wiki/Script

[6] DR. Stinson and R. Strobl, "Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates http://cacr.uwaterloo.ca/techreports/2001/corr2001-13.ps, 2001

[7] "Verifiable secret sharing" https://en.wikipedia.org/wiki/Verifiable_secret_sharing https://ja.overleaf.com/project/606fb0ab21f88266

[8] Tapyrus Signer, https://github.com/chaintope/tapyrus-signer

[9] Tapyrus Core, https://github.com/azuchi/tapyrus-core

[10] Bitcoin Core, https://github.com/bitcoin/bitcoin/

[11] Thaddeus Dryja, Joseph Poon, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments" d http://lightning.network/lightning-network-paper.pdf, 2016

[12] Andrew Poelstra, "Mimblewimble and scriptless scripts" http://diyhpl.us/wiki/transcripts/realworldcrypto/2018/ mimblewimble-and-scriptless-scripts/, 2018

[13] Flavien Charlon, "Open Assets Protocol" https://github.com/OpenAssets/open-assets-protocol, 2014

[14] Counterparty Protocol https://github.com/CounterpartyXCP/Documentation/blob/ master/Developers/protocol_specification.md

[15] "EIP-20: ERC-20 Token Standard", https://eips.ethereum.org/EIPS/eip-20

[16] Compact Size, https://en.bitcoin.it/wiki/Protocol_documentation#Variable_length_integer

[17] Dan Boneh, Benedikt B ¨ unz, Ben Fisch, Stanford University
"Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains"
https://eprint.iacr.org/2018/1188.pdf