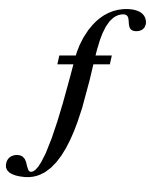


Flame

Decentralized electronic cash with confidentiality and smart contracts

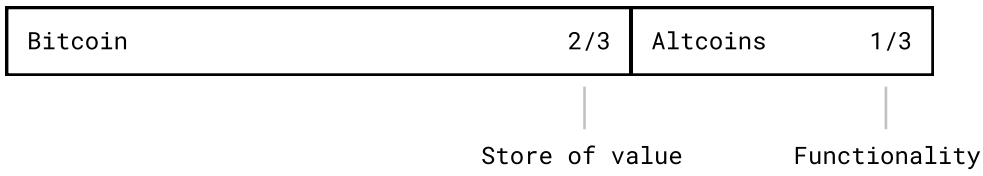
Oleg Andreev · oleg@safely.app

Abstract. Flame is an electronic cash network based on Bitcoin. Flame introduces decentralized proof-of-burn consensus with a unified architecture that combines confidentiality and a powerful programming environment. The network continuously burns bitcoins as a means to timestamp blocks of transactions. As an incentive, the network issues new units of *flames*.

1. Introduction

In 2009, Bitcoin [BTC] presented a novel approach to peer-to-peer cryptographic money. Bitcoin allowed anyone to participate in a *proof-of-work* consensus protocol that served two purposes: securing transactions against double-spending in a trust-minimized manner, and producing and distributing units of a new kind of collectible: *bitcoins*.

Over time, the cryptocurrency market took a distinctive shape: as of 2026, approximately two-thirds of total cryptocurrency market capitalization is attributed to Bitcoin [CMC] as a value asset, and one-third to altcoins. In terms of functionality, Bitcoin is inferior to the majority of the altcoins: it has high confirmation latency, low bandwidth, limited programmability and no on-chain privacy. Yet, the majority of the market value is allocated to this *valuable asset*, while only a third is allocated to *functionality* for payments and trading.



This functionality is fragmented and disconnected from a truly valuable asset: every project optimizes its own “killer feature” with significant engineering trade-offs — centrally allocated tokens, semi-trusted bridged assets or stablecoins backed by conventional finance.

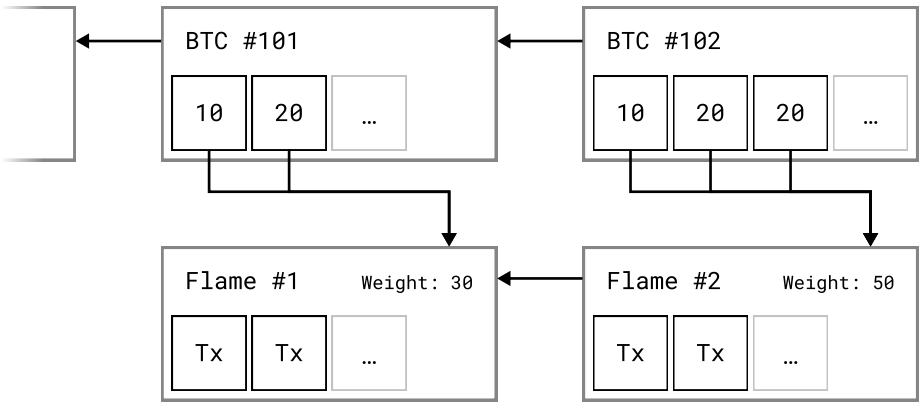
In this paper, we present Flame: a peer-to-peer network that solves the double-spending problem through bitcoin-based proof-of-burn consensus and provides a powerful programming architecture for confidential, high-performance financial applications.

2. Proof-of-burn consensus

Satoshi Nakamoto proposed [BTC] a proof-of-work mechanism to solve the double-spending problem in a decentralized manner. Years later, many more altcoins experimented with a variety of Byzantine agreement protocols: alternative proof-of-work algorithms, merged mining, variations of proof-of-stake and BFT schemes. To date, no altcoin project has approached Bitcoin's level of security, as they all suffer from inherent centralization and censorship risks [PoS1, PoS2, MM, BCH].

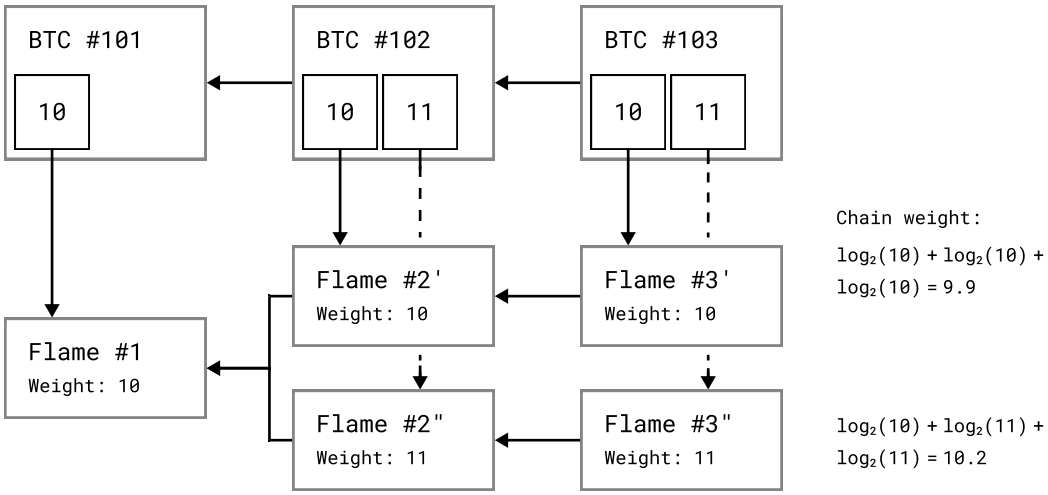
Flame uses a proof-of-burn mechanism for solving the double-spending problem, serving the same role as proof-of-work does in Bitcoin. Nodes that participate in the protocol, called *minters*, verify blocks of transactions according to the network's rules and continuously burn bitcoins as part of the agreement protocol to select a common chain among multiple possible valid ones. Bitcoin provides the record and ordering of burns and endorsements, while Flame's fork-choice rule determines which valid history to follow.

Proof-of-burn schemes were first sketched by dacoinminster and Iain Stewart [PoB]; Slimcoin [SM] then implemented a variant of proof-of-burn using *its own* coins, raising concerns similar to those of a proof-of-stake system. Flame is the first attempt to implement a consensus protocol based on continuous burning of bitcoins.



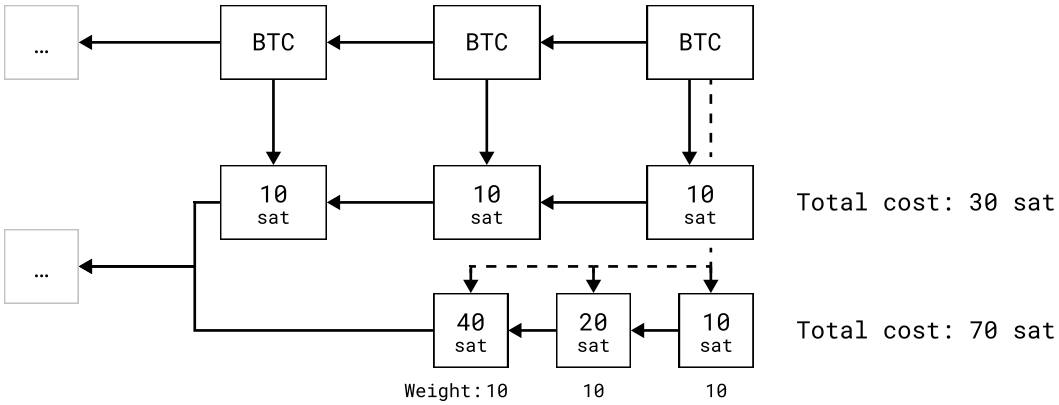
Flame chain with proof-of-burn weights equal to amounts of burned bitcoins.

As in a proof-of-work network, it is possible that two or more Flame blocks at the same height are endorsed by proof-of-burn. In that case, nodes choose the chain with higher cumulative proof, but retain the alternative branches in case they become stronger in the future.



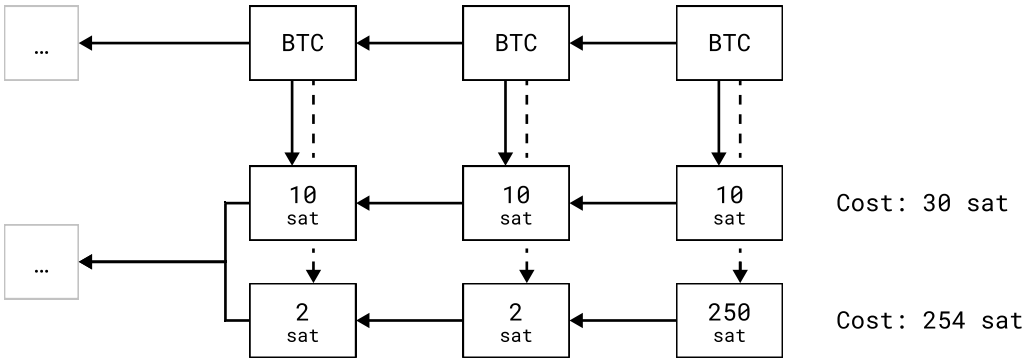
Choosing the chain with the most weight.

The weight of each Flame block is determined by the weights of its endorsements, which are defined by the amount of bitcoin burned. Delayed endorsements, where Flame block N is endorsed at Bitcoin block N + k, are exponentially discounted to mitigate attempts to fork the chain at a later date. Delays are nonetheless allowed to enable Flame to survive Bitcoin chain reorganizations.



Delayed Flame blocks get exponentially discounted weight.

The total weight of the chain is computed as the sum of binary logarithms of individual block weights. This measures sustained support rather than total expenditure of bitcoins: over a fixed number of blocks, a given total effective burn produces the greatest chain weight when distributed evenly. Concentrating expenditure in one block is therefore less effective than supporting successive blocks.



Concentrating weight in the last block becomes exponentially more expensive.

Flame allows for reorganizations at two levels: at the Bitcoin level and at the Flame level. Nodes first switch to the strongest Bitcoin branch to count endorsements, then adjust weights and select the strongest Flame branch.

3. Incentive

Minters are rewarded for their contribution with *flames*. Flames are issued automatically according to a fixed schedule with a limit on total supply of 21 million units. In each block, an allotment of flames is distributed among minters in proportion to their contributions to the proof-of-burn. One *flame* is subdivided into 100 million subunits called *sparks*.

Flame aims to reproduce Bitcoin's economic protection against double-spending through continuous, irreversible expenditure. Assuming that ordinary participation supplies a steady rate of burn, a competing history must attract a comparable expenditure over the same period. An attacker that falls behind and attempts to recover its deficit in only a few blocks faces a premium from logarithmic weighting; endorsements included late in Bitcoin incur an additional exponential discount. Rewards on the accepted chain are intended to encourage continuous participation, while the cost of replacing its history depends on the distribution and timing of the competing endorsements.

4. Flame Virtual Machine

Bitcoin pioneered the implementation of smart contracts with its “script” mechanism. Bitcoin scripts act as “predicates” that control access to individual coins. Bitcoin transactions contain inputs and outputs, where inputs satisfy one or more such predicates with signatures and other conditions, such as witness data and time-locks, unlock coins and then merge or split them towards new outputs with a predicate per output. This is known as the “UTXO model” (“unspent transaction outputs”). Such an approach proved to be limited, but scalable and robust: every user has to take care of their own coins; transactions can be processed in parallel.

Ethereum [ETH] proposed an even more powerful “actor model” (also known as the “account model”), where an “actor” is a program with its own isolated state. Actors receive incoming messages from the external world, such as transfer instructions signed by users, and from other actors. In this way, the whole network simulates a decentralized “computer cluster” consisting of individual isolated “computers”. This model allowed the creation of a “decentralized finance” industry. The downside is poor scalability: transactions have to be strictly ordered and storage has to be managed with extra complexity to mitigate denial-of-service risks. Telegram Open Network [TON] proposed a more scalable approach to the actor model built around proof-of-stake consensus.

To achieve storage scalability, confidentiality and rich programmability, Flame combines both approaches within a single execution environment, the Flame Virtual Machine. FlameVM follows in the footsteps of Chain’s TxVM [TxVM] and Interstellar’s ZkVM [ZkVM], which introduced direct manipulation of linear types and confidential expressions, and goes further with support for *actors* and *messages*.

Type system

FlameVM combines linear types [LL] with capability-based [OC] control of assets. Tokens and contracts are native, move-only values: programs cannot duplicate them or silently discard the assets they contain. Conservation of value during transfers is enforced by the VM, rather than reimplemented through application bookkeeping.

Contract predicates execute in isolated frames, receiving their payload and explicitly supplied arguments without inheriting the calling actor’s storage access or authority. This helps prevent failures caused by untrusted contract code exercising its caller’s privileges, such as “confused-deputy” bugs. Together, these primitives make asset movement and authority boundaries explicit, allowing financial agreements to be expressed directly in the programming model.

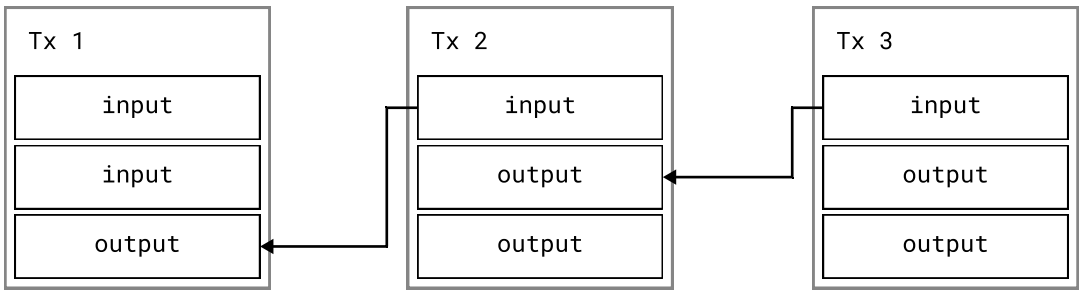
Unlike most blockchain scripting environments, FlameVM is designed with a higher-level type system. Apart from tokens, the VM provides users with scalars, elliptic curve points, dictionaries, cells, expressions with confidential data and batched multi-scalar cryptographic operations for efficient custom zero-knowledge proofs. FlameVM comes with "batteries included": users do not have to build necessary abstractions such as linear logic and containers using the low-level primitives and idiosyncrasies of the protocol. Instead, the VM's core includes all the tools necessary to build financial applications.

External transactions and contracts

A FlameVM *external transaction* is a data structure around a program that describes the issuance and transfer of *tokens* and other *linear types*. A transaction also contains additional "witness" data, such as cryptographic proofs necessary to verify the correctness of the transaction. External transactions are signed by their respective authors, broadcast by nodes and included in blocks.

Tokens are created via *programmatic issuance*: each token *flavor* is securely identified by its issuer's program, which specifies rules authorizing the creation of additional units. *Flames* are ordinary tokens with a designated flavor that are issued automatically according to the proof-of-burn consensus rules.

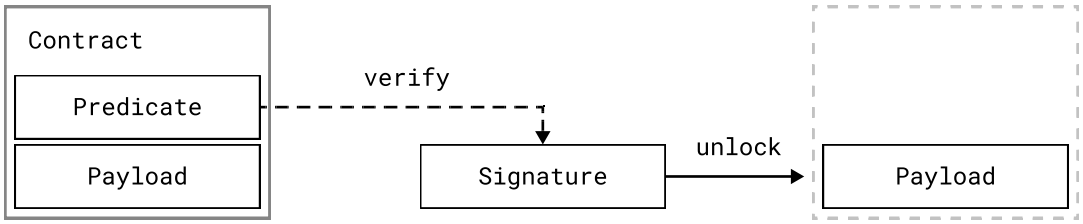
Tokens are transferred through inputs and outputs. Each output specifies a new destination for the funds, and each input identifies an output from a previous transaction and unlocks its value. The input is said to spend the earlier transaction's output.



Value flow from inputs to outputs in external transactions.

Unspent outputs are called contracts. Contracts protect their payloads (tokens and data) with a predicate using a variant of the Taproot scheme [TR]. Predicates can be used directly as public keys to verify the transaction signature, but may contain a more sophisticated set of conditions: they can pack multiple public keys for a multi-signature policy or arbitrarily complex programs.

When the contract conditions are satisfied, the contract is destroyed and its contents are available to the transaction for subsequent use.



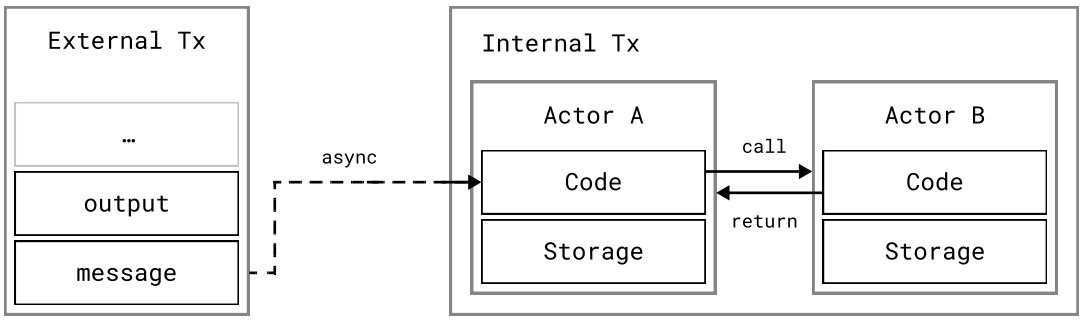
Contracts implement the object capability model by guarding payloads with predicates.

Contracts are most suitable for executing private few-party agreements: nodes do not have to pay for storage and contracts can easily express conditions on confidential tokens and data. However, contracts do not allow access in random order, which is necessary for multi-player contracts such as decentralized exchanges. To build these, Flame offers actors.

Internal transactions and actors

Actors are instances of code and storage, similar to contracts, but each with a persistent address accessible by any user. An actor's storage is modifiable in place, while contracts are destroyed after each use. Actors require network nodes to store their data verbatim and therefore must pay for annual storage leases.

To interact with actors, external transactions produce *messages* in addition to outputs. Messages accept the same linear types as contract payloads. Messages are FlameVM values that get processed by an actor's code in *internal transactions*. Messages are processed asynchronously: the external transaction is included in the block regardless of the result of the internal transaction.



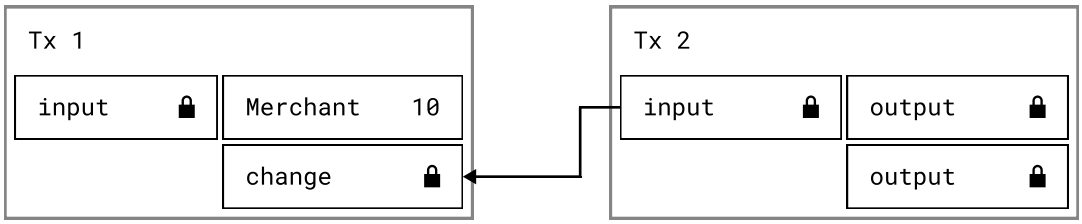
Actors receive messages from external transactions.

Apart from sending messages, actors can communicate *synchronously* by making calls to each other. All the effects of such nested calls are recorded atomically as a single internal transaction.

To keep track of resource usage within all kinds of transactions, FlameVM assigns a *gas cost* to every operation. All transactions are executed within a declared gas budget that is paid for via a *transaction fee*. Actors also have to pay for storage via annual leases, as described in §6, Storage.

5. Confidentiality

In Flame transactions, all quantities and flavors are hidden by default using cryptographic commitments. Contracts may also keep arbitrary numeric data hidden. At the moment of issuance, an asset type must be public because the flavor is tied to the issuance program, but the quantity could remain hidden.



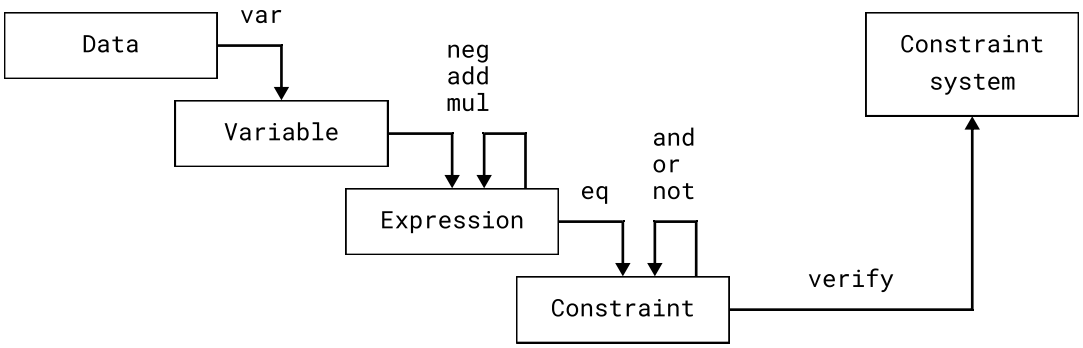
Amounts remain hidden from the network; the merchant sees its payment, but no balances or amounts before or after it.

The graph of transactions remains public to efficiently prevent double-spending. Common graph-hiding designs retain spent identifiers to prevent double-spending, alongside commitments to outputs. This creates state that grows with transaction history, while techniques for limiting that growth affect the security model and smart contract capabilities. Flame makes a different trade-off: explicitly tracking unspent outputs allows storage to be dramatically optimized and smart contracts to operate freely on hidden values.

Non-trivial contracts can keep their parameters, such as prices, expiration times and counters, hidden at a low cost and verify constraints on private values without revealing those values. Predicates use the Taproot scheme to pack an arbitrarily large set of conditions into a single public key. This allows users to avoid revealing contract logic in the first place.

If all parties to a contract cooperate, for example by closing a payment channel normally without forced settlement, the network only has to see an aggregated signature, so neither the number of parties nor the nature of the contract has to be published. If the parties do not cooperate, Taproot allows a required subset of conditions to be revealed and verified while keeping all the others secret.

The programmable constraint system, pioneered in Interstellar's ZkVM and available within external transactions, is based on Bulletproofs [BP], Ristretto [RT] and Merlin [MR]. FlameVM provides types for arithmetic and logical expressions on hidden values that become part of a single arithmetic circuit subject to a single zero-knowledge proof. Every secret value starts as a Pedersen commitment [PC] and becomes part of one or more linear expressions. These expressions are combined into Boolean constraints, which together form the dynamically constructed constraint system.



Lifecycle of arithmetic expressions and logical constraints with hidden variables.

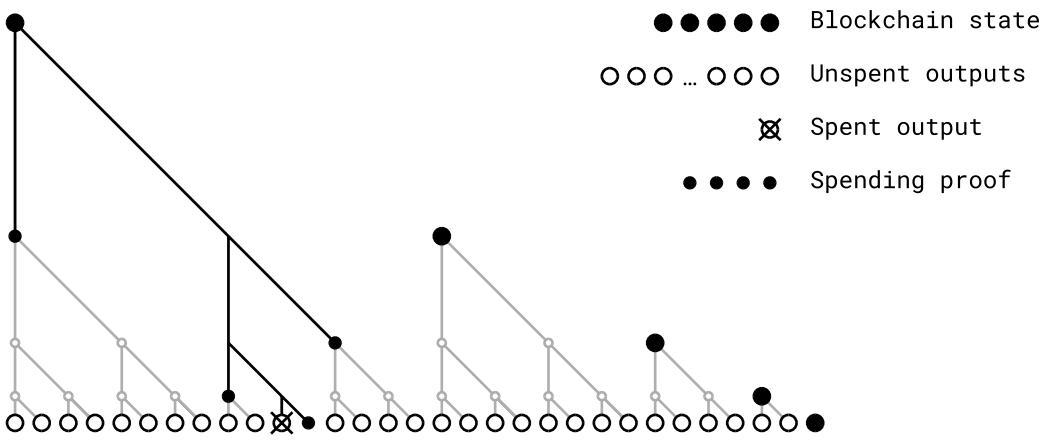
FlameVM also provides lower-level types for scalars, group elements, transcripts and batched multi-scalar multiplication for building custom ZK proofs and interoperating with Bulletproofs. This tooling makes ZK proofs possible inside actors as well, allowing users to verify custom proofs and bind them to arbitrary data inside actor-based applications.

6. Storage

Flame, like Bitcoin, replicates validation across nodes rather than dividing the ledger among them. Each node independently processes the network's transactions and has to store all the data necessary to operate the network. Flame addresses storage scaling with a *Utreexo accumulator* and *leased storage*.

Balances held in contracts are represented by unspent transaction outputs (UTXOs). The set of UTXOs is in turn compressed using the Utreexo protocol proposed by Thaddeus Dryja [Utreexo].

Utreexo allows nodes to keep only the roots of perfect Merkle trees and lets users keep track of their own UTXOs within such a “forest”. Every input in an external transaction provides a Merkle proof of inclusion in the Utreexo accumulator that acts simultaneously as a proof and as the missing data needed to update the structure for the next block. Utreexo fully externalizes the cost of storing private data to users.



Utreexo compresses the storage of all unspent outputs into a short list of Merkle roots.

Actors use a storage mechanism separate from Utreexo to enable non-exclusive access to their state by all users via asynchronous messages. To protect nodes from exhausting their persistent storage space, the network limits the amount of storage available, issues new bytes at a constant rate and regulates the price of annual leases according to the consumption of the remaining storage space. As free space gets smaller, the price increases geometrically and vice versa. When one or more leases expire, leaving no room for an actor’s storage, the actor is frozen and only a single hash of its state is kept by the nodes. If the actor is not abandoned, users who keep track of its state may deliver the missing data in an external transaction and purchase a new storage lease. Storage released by expired leases returns to the available pool and can be purchased by any actor again.

All data in Flame is serialized using *cells* modeled on the TVM Cells proposed by Dr. Nikolai Durov [TVM]. Cells offer a prunable encoding format for contracts, predicates and actors, and implement Merkle proofs [MP]. Each cell contains a short string of bytes and up to four references to nested cells, thereby forming a directed acyclic graph. Cell references can be pruned and replaced with their hashes. The pruned data can then be provided in full or in part as a connected subset of the graph down to the required cell. Such a partial opening of a pruned cell constitutes a *Merkle proof*. Cells provide a uniform method to store, compress and uncompress data throughout the protocol, compute identifiers for transactions, contracts and actors, decode contracts from UTXOs, open Taproot predicate programs, unfreeze actors with expired storage leases and programmatically compress the state of contracts and actors to save space.

To summarize, Flame provides scalable private storage for users' balances and data. For multi-party coordination via actors, there is limited storage available for lease that is guaranteed to be stored by nodes. The combination of the two kinds of storage allows for efficient implementations of massive multi-user applications: for example, a decentralized exchange may use a limited amount of leased storage for its liquidity pools, while any number of users may hold shares of these pools as tokens inside contracts compressed via Utreexo.

7. Network

The Flame network consists of nodes, some of them operating as minters (analogous to miners) who create blocks of transactions, and others merely validating blocks and receiving payments. The steps to run the network are as follows:

1. Each minter starts by placing a *stake* on the Bitcoin network by publishing a transaction that permanently locks up (*burns*) some amount of bitcoin and declares the minter's public key for a minting period.
2. Nodes broadcast external transactions.
3. Minters use a BFT protocol [PBFT] to build an intermediate chain of *iblocks* (intermediate blocks) as they synchronize the broadcast transactions among themselves. The quorum is defined by the set of active stakes at the latest Bitcoin block.
4. After every Bitcoin block N, the BFT quorum produces a special iblock as the candidate for Flame block N + 1.
5. After the minters agree on Flame block N + 1, each minter broadcasts a Bitcoin transaction announcing its *endorsement* for that Flame block using its stake, weighted according to the size of the *stake*.
6. The sum of all endorsements' weights constitutes the proof-of-burn for the Flame block.
7. If a node encounters two valid chains, it keeps both of them and switches to the one with the largest total weight, computed as a sum of logarithms of block weights.
8. In the event of a Bitcoin reorganization (a change of the main chain), minters first switch to the Bitcoin chain, re-evaluate the weights of all endorsements of all known valid Flame chains and then perform a reorganization of the Flame chain.

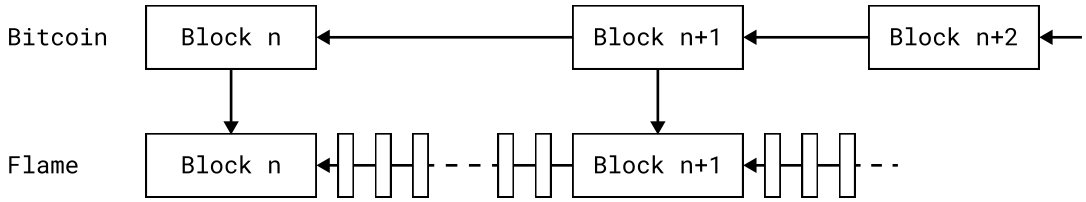
Burning coins in advance for several blocks increases the cost of attacks on the network, minimizes the amount of bitcoin controlled by online keys and enables delegated minting.

Each stake can contribute weight only once per Flame height: nodes count its first valid endorsement in Bitcoin-chain order and ignore subsequent endorsements for the same height.

Note that minters use BFT pre-agreement as a method to effectively utilize their stakes, not as the ultimate source of truth.

8. Fast payment verification

Low-latency payment confirmation is possible using BFT pre-agreement. Minters continuously synchronize their sets of transactions and timestamp them as intermediate blocks (iblocks). By the time a Flame block needs to be produced, most transactions have been processed and the latest iblock can be endorsed via proof-of-burn. Pre-agreement may be used to provide payment assurance without waiting for the next Bitcoin block.



Intermediate blocks allow low-latency confirmation for low-risk payments.

Minters publish verifiable evidence of pre-agreement on a chain of intermediate blocks. This lets recipients verify that a payment belongs to the history participating minters have agreed to endorse, before those endorsements appear in Bitcoin. Provided minters remain coordinated and continue supporting that history, dissenters must support an alternative with enough effective proof-of-burn to prevail. Pre-agreement therefore provides early assurance of inclusion, not a separate source of finality: recipients may accept this additional risk for smaller payments, while payments requiring stronger assurance wait for Bitcoin-recorded endorsements and subsequent Flame confirmations.

9. The future

Flame expands the utility of bitcoins with confidentiality, programmability and scalable storage without requiring changes to Bitcoin's consensus rules. By participating in the proof-of-burn protocol, nodes perform a one-way conversion of bitcoins into flames. While the issuance schedule of flames is fixed, competition determines how much BTC participants are willing to burn to obtain a share of that issuance, thus forming a dynamic bitcoin-to-flame *mint rate*. As a result, the market determines which portion of bitcoins is permanently transitioned into the new network as flames.

Bitcoin's security budget depends on the mix of inflation (emission of newly mined bitcoins) and transaction fees. For the time being, inflation is covering the majority of mining costs. However, when inflation is halved a few more times, transaction fees will start to define Bitcoin's security budget. To keep the security budget healthy, at least one of three conditions must be satisfied: (1) transaction volume increases, (2) fees grow, (3) the purchasing power of bitcoin rewards remains high.

Flame can contribute to all of the above: minting causes continuous demand for space in Bitcoin blocks; endorsement transactions pay priority fees to avoid delays; burns reduce spendable supply and may increase the purchasing power of Bitcoin's mining rewards, without paying miners directly or changing Bitcoin's issuance schedule.

Bitcoin miners may find it beneficial to strengthen Flame's security by rejecting double-spend attempts as they independently follow the Flame protocol. Improved security of Flame may contribute to its market value, which can encourage minters to burn more bitcoins.

Glossary

Miner	a Bitcoin node that participates in proof-of-work consensus.
Minter	a Flame node that participates in proof-of-burn consensus.
Burn	a Bitcoin transaction that permanently locks up bitcoins.
Stake	a burn that declares the minter's identity.
Endorsement	a Bitcoin transaction that associates a stake with a particular Flame block.
Pre-agreement	verifiable agreement among participating minters on a candidate chain of iblocks.
Candidate block	a result of pre-agreement that can receive endorsements.
Flame block	part of the Flame chain that has weight due to endorsements.
Flame iblock	an intermediate block produced by pre-agreement.
Flame (currency)	a unit of the native token issued according to the consensus rules.
Spark	the indivisible unit of the incentive within the consensus protocol; one hundred-millionth of a flame.
Flavor	a unique cryptographic identifier of a class of fungible tokens, represented by a scalar.
Quantity	the amount of a token of a certain flavor, represented by a 64-bit unsigned integer.
Linear type	a data type that can be moved, created or destroyed within invariants defined by the consensus rules.
Token	a linear type that can be issued and transferred.
Contract	a linear type that can be used once, containing a payload protected by a predicate.
Payload	a data type that is stored inside a contract, a message or an actor.
Predicate	a condition for accessing a contract's payload, compressed into a point.
Public key	a point used to authorize a transaction via a Schnorr signature.
Actor	a program with persistent storage and identity that processes messages and pays for storage leases.
Message	an effect of a transaction that asynchronously delivers a payload to an actor.
Scalar	an integer modulo 2^{252} + 27742317777372353535851937790883648493 (the Ristretto group order).
Point	an element of the Ristretto group used to represent a public key, a commitment or a predicate.
Ristretto	a prime-order group based on Curve25519.

Commitment	a Pedersen commitment to a value v in the form v•G + f-H, where G is the primary base point, H is an independently derived generator whose discrete-log relation to G is assumed unknown, and f is a blinding scalar.
Variable	a secret value within a Bulletproofs constraint system.
Expression	a linear combination of variables.
Constraint	a Boolean function of expressions and constraints.
Cell	a primitive hierarchical data structure for storing byte strings and nested cells.
Merkle proof	data sufficient to verify that an element belongs to a hash-committed structure without revealing the entire structure (such as a subset of cells).
Transaction	a record of changes to the network state, which may be external or internal.
External transaction	a signed data structure that specifies transfers of tokens from inputs to outputs and sends messages to actors.
Internal transaction	a record of updates within actors caused by a message.
Input	the act of spending an output.
Output	a contract available for a later spend.
UTXO	an unspent transaction output.
Utreexo	a hash-based accumulator that compactly represents the UTXO set and supports verification using membership proofs.
Lease	a portion of actor storage allocated for one year (52,500 blocks) to an actor for its payload.

References

[BTC] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, bitcoin.org/bitcoin.pdf, 2008.

[CMC] CoinMarketCap, Bitcoin Dominance, coinmarketcap.com/charts/bitcoin-dominance, September 30, 2026.

[PoS1] Andrew Poelstra, On Stake and Consensus, download.wpsoftware.net/bitcoin/pos.pdf, 2015.

[PoS2] Eric Voskuil, Proof of Stake Fallacy (in Cryptoeconomics: Fundamental Principles of Bitcoin), voskuil.org/cryptoeconomics/cryptoeconomics.pdf, 2020.

[MM] Aljosha Judmayer, Alexei Zamyatin, Nicholas Stifter, Artemios Voyiatzis, Edgar Weippl, Merged Mining: Curse or Cure?, eprint.iacr.org/2017/791.pdf, 2017.

[BCH] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, Yongdae Kim, Bitcoin vs. Bitcoin Cash: Coexistence or Downfall of Bitcoin Cash?, arxiv.org/abs/1902.11064, 2019.

[PoB] Iain Stewart, Proof of burn: a potential alternative to proof of work and proof of stake, bitcointalk.org/index.php?topic=131139.0, 2012.

[SM] P4Titan, Slimcoin: A Peer-to-Peer Crypto-Currency with Proof-of-Burn, slimcoin-project.github.io/whitepaperSLM.pdf, 2014.

[ETH] Vitalik Buterin, Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, ethereum.org/en/whitepaper/, 2014.

[TON] Nikolai Durov, The Open Network, docs.ton.org/ton.pdf, 2021.

[TxVM] Bob Glickstein, Cathie Yun, Dan Robinson, Keith Rarick, Oleg Andreev, TxVM: A New Design for Blockchain Transactions, github.com/chain/txvm/blob/main/whitepaper/whitepaper.pdf, 2018.

[ZkVM] Oleg Andreev, Bob Glickstein, Vicki Niu, Tess Rinearson, Debnil Sur, Cathie Yun, ZkVM: fast, private, flexible blockchain contracts, github.com/stellar/slingshot/files/3164245/zkvm-whitepaper-2019-05-09.pdf, 2019.

[LL] Jean-Yves Girard, Linear logic, doi.org/10.1016/0304-3975(87)90045-4, 1987.

[OC] Mark S. Miller, Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, erights.org/talks/thesis/, 2006.

[TR] Pieter Wuille, Jonas Nick, Anthony Towns, Taproot: SegWit version 1 spending rules, github.com/bitcoin/bips/blob/master/bip-0341.mediawiki, 2020.

[BP] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, Greg Maxwell, Bulletproofs: Short Proofs for Confidential Transactions and More, eprint.iacr.org/2017/1066, 2017.

[RT] Henry de Valence, Jack Grigg, Mike Hamburg, Isis Lovecruft, George Tankersley, Filippo Valsorda, The ristretto255 and decaf448 Groups, www.rfc-editor.org/rfc/rfc9496.html, 2023.

Acknowledgements

The work is based on extensive research by the Bitcoin developer community: Pieter Wuille, Gregory Maxwell, Thaddeus Dryja, Andrew Poelstra and many others.

FlameVM is an evolution of ZkVM and TxVM, authored by Oleg Andreev, Dan Robinson, Bob Glickstein, Henry de Valence, Cathie Yun, Vicki Niu, Tess Rinearson and Debnil Sur at Chain Inc. and Interstellar Inc.