A Softly Technical Introduction of Bitlayer's Bitcoin Taptree STARK

Bitlayer Research Team

December, 2024

Abstract: Bitlayer introduces a Turing-complete computational layer built upon the BitVM paradigm and a novel zero-knowledge proof (ZKP) protocol designed to fully inherit Bitcoin's security model. This proposed ZKP, termed *Bitcoin Taptree STARK* (TapSTARK), is a succinct and transparent argument of knowledge adapted from the STARK protocol. In this blog, we provide a concise and accessible technical introduction to TapSTARK.

1. Introduction

We first introduce some background knowledge required for this blog, including ZKP (Zero-Knowledge Proof), Polynomial Commitment, and Merkle Tree.

1.1 Zero-knowledge proof

ZKP allows a prover to convince a verifier of a statement's validity without revealing additional information. A *succinct* ZKP ensures sublinear communication relative to the statement size, while a *transparent* ZKP avoids trusted setups, making it blockchain-compatible. TapSTARK does not require the zero-knowledge property.

In (Tap-)STARK, ZKPs use Algebraic Intermediate Representations (AIRs) to encode computations into polynomial constraints with degree bounds. For example, to prove a d-degree polynomial $f(X)$ evaluates to 0 over a size-m subgroup $H = (1, \omega, \ldots, \omega^{m-1})$, one shows

$$f(X) = g(X) \cdot Z_H(X),$$

where $Z_H(X) = (X-1)(X-\omega)\cdots(X-\omega^{m-1})$ vanishes on H.

Directly proving this involves sending the coefficients of $f(X)$ and $g(X)$, resulting in proof size and verifier complexity linear to d. Using the Schwartz-Zippel Lemma, the verifier instead selects a random r, checks $f(r) = g(r) \cdot Z_H(r)$, and concludes correctness with high probability. This reduces proof size and complexity to sublinear in d, as only evaluations are exchanged. Furthermore, $Z_H(X)$ evaluations take $O(\log(d-m))$ time since $Z_H(X)$ simplifies to $X^m - 1$ for H. Without safeguards, however, a prover could send falsified $f^*(r)$ and $g^*(r)$ satisfying $f^*(r) = g^*(r) \cdot Z_H(r)$.

1.2 Polynomial Commitment

(Tap-)STARK employs a cryptographic primitive called a *polynomial commitment scheme* (PCS) to ensure the validity of $f(r)$ and $g(r)$. A PCS enables a prover to commit to a polynomial $f(X)$ over a field $\mathbb{F}$ with degree bound d and convince a verifier (via an Eval protocol) that $y = f(x)$ for public $x, y \in \mathbb{F}$, even when the verifier selects x after the commitment is made.

The PCS is defined through the following algorithms:

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, d)$: Generates the public parameter pp from the security parameter λ and degree bound d.
- $C \leftarrow \mathsf{Com}(\mathsf{pp}, f)$: Computes the commitment C for polynomial f using pp.
- $b \leftarrow \mathsf{Eval}(\mathsf{pp}, C, x, y, f)$: An interactive argument of knowledge between a probabilistic polynomial time prover $\mathcal{P}$ and verifier $\mathcal{V}$, denoted $b \leftarrow \langle \mathsf{Open}(f), \mathsf{Verify}\rangle(\mathsf{pp}, C, x, y)$. The prover $\mathcal{P}$ convinces $\mathcal{V}$ that $f(x) = y$, and $\mathcal{V}$ outputs either accept or reject.

With PCS, a prover can validate an AIR by committing to the secret polynomials and sending these commitments to the verifier. The verifier then selects a random r, and the prover and verifier use the Eval protocol to verify polynomial evaluations at r. Finally, the verifier checks the polynomial constraints and decides to accept or reject.

1.3 Merkle Tree

A Merkle tree is a vector commitment scheme with linear prover complexity and logarithmic verifier complexity and proof size, all proportional to the vector size. It consists of the following algorithms:

- $\mathsf{rt} \leftarrow \mathsf{MT.Com}(\boldsymbol{v})$: Computes the Merkle tree root rt for the vector $\boldsymbol{v}$.
- $(\{v_i\}_{i\in\mathcal{I}}, \mathsf{path}) \leftarrow \mathsf{MT.Open}(\mathcal{I}, \boldsymbol{v})$: Outputs the queried values $\{v_i\}_{i\in\mathcal{I}}$ and the corresponding verification path path for query indices $\mathcal{I}$.
- $\mathsf{MT.Verify}(\mathsf{rt}, \mathcal{I}, \{v_i\}_{i\in\mathcal{I}}, \mathsf{path})$: Verifies correctness of the queried values against the root rt using the path path.

2. Overview

At a high level, TapSTARK builds upon STARK by inheriting its AIR phase, which transforms the original constraints into low-degree polynomial constraints. The primary difference lies in the use of a novel Bitcoin Taptree FRI (TapFRI) as its PCS. Thus, TapSTARK can be expressed as:

$$\text{TapSTARK} = \text{AIR} + \text{TapFRI}.$$

TapFRI replaces the standard Merkle tree in FRI PCS with Bitcoin's native Merkle tree, Taptree. This substitution enables gas-free verification of Merkle paths, as Bitcoin's consensus mechanism handles the verification. Additionally, Taptrees eliminate the need to construct Merkle trees on-chain, thereby avoiding reliance on non-standard instructions like OP-CAT for string concatenation.

However, integrating FRI PCS with Taptree is not straightforward due to several challenges:

1. **Commitment Representation:** Taptree is a Merkle-style hash tree where each leaf is a Bitcoin script rather than a field element, as required in FRI PCS. This raises the question of how to commit field elements using Bitcoin scripts.
2. **Script Consistency:** Bitcoin scripts are limited to a maximum size of 400 KB, and a single field element may be referenced across multiple scripts. Ensuring consistency for such elements across different scripts is non-trivial.
3. **Verifier Flexibility:** Unlike the programmable verifier in standard FRI PCS, Bitcoin script verification is fixed and lacks flexibility.

Before introducing the specific techniques in TapFRI and TapSTARK, we first outline the operation of the FRI PCS.

TapSTARK utilizes a PCS based on the *fast Reed-Solomon interactive oracle proof of proximity* (FRI). Given a degree bound d and a committed vector, FRI enables a prover to convince a verifier that the committed vector is δ-close to $\mathrm{RS}[L, d/|L|]$ in terms of relative Hamming distance.

The prover's complexity is $O(d)$, while the verifier's complexity and proof size are $O(\log^2 d)$. With a suitably small δ, FRI effectively performs a low-degree test, ensuring the prover knows a *unique* polynomial $f(X)$ corresponding to the committed vector with $\deg(f) \leq d$. FRI PCS is constructed by combining FRI with a remainder argument: if $f(z) = y$ for a degree-d polynomial $f(X)$, then $(f(X) - y)/(X - z)$ is a degree-$(d-1)$ polynomial. In FRI PCS, vector commitments are implemented using Merkle trees.

For polynomials $f_1, \ldots, f_t$ with degree bounds $d_1, \ldots, d_t$, *batch-FRI* can verify that all $f_1, \ldots, f_t$ are valid low-degree polynomials. Specifically, the prover and verifier use FRI to test whether $f|_L \in \mathrm{RS}[L, d_{\max}/|L|]$ for

$$f(X) = \sum_{i=1}^{t} \lambda^{i-1} \cdot f_i(X) \cdot X^{d_{\max} - d_i},$$

where $d_{\max} = \max\{d_1, \ldots, d_t\}$, and λ is chosen randomly by the verifier. The FRI PCS is directly derived from batch-FRI, as described in Algorithm 1.

Algorithm 1: The FRI PCS. A polynomial f with a degree bound d and $\mathsf{pp} = \{\mathbb{F}, L\} \leftarrow \mathsf{Gen}(1^\lambda, d)$, where $\mathbb{F}$ is a finite field and L is a multiplicative coset of size $O(d)$. WLOG, assume d is a power-of-two integer.

```
• $C \leftarrow \mathsf{Com}(\mathsf{pp}, f)$: $\mathcal{P}$ sets $L_0 = L$, computes $f|_{L_0}$, and generates
  $C \leftarrow \mathsf{MT.Com}(f|_{L_0})$.
• $b \leftarrow \langle \mathsf{Open}(f), \mathsf{Verify} \rangle(\mathsf{pp}, C, z, y)$: The prover $\mathcal{P}$ holds a polynomial $\mathcal{P}$
  and the verifier $\mathcal{V}$ know $d, z, y$. To convince $\mathcal{V}$ that $f(z) = y$, $\mathcal{P}$ and $\mathcal{V}$:
    1. $\mathcal{V}$ sends a random $\lambda$ to $\mathcal{P}$.
    2. $\mathcal{P}$ sets $f_0(X) = f(X) + \lambda \cdot X \cdot \frac{f(X)-y}{X-z}$.
    3. For $i = 1$ to $\mathsf{r}+1$, where $\mathsf{r} = \log_2 d$:
       (a) $\mathcal{P}$: decomposes $f_{i-1}$ as $g_i(X^2) + X \cdot h_i(X^2)$.
       (b) $\mathcal{V} \to \mathcal{P}$: $\alpha_i \leftarrow_{\$} \mathbb{F}$.
       (c) $\mathcal{P}$: computes $f_i(X) \leftarrow g_i(X) + \alpha_i \cdot h_i(X)$. If $i \neq \mathsf{r}+1$, compute
           $f_i|_{L_i}$ for $L_i = \{x^2 | \, x \in L_{i-1}\}$, run $\mathsf{rt}_i \leftarrow \mathsf{MT.Com}(f_i|_{L_i})$ and
           send $\mathsf{rt}_i$ to $\mathcal{V}$; Else, send $f_i$ to $\mathcal{V}$.
    4. For $j = 1$ to $q = O(\lambda)$:
       (a) $\mathcal{V} \to \mathcal{P}$: $\beta_j \leftarrow_{\$} L_0$.
       (b) For $i = 1$ to $\mathsf{r}+1$:
           i. $\mathcal{P}$: invokes $\mathsf{MT.Open}$ to open $f_{i-1}(\pm\beta^{2^{i-1}}), f_i(\beta^{2^i})$.
           ii. $\mathcal{V}$: checks the correctness of related Merkle trees by
               $\mathsf{MT.Verify}$. Check whether the three pairs

               $(\beta^{2^{i-1}}, f_{i-1}(\beta^{2^{i-1}})), (-\beta^{2^{i-1}}, -f_{i-1}(\beta^{2^{i-1}})), (\alpha_i, f_i(\beta^{2^i}))$

               are on a common degree-1 polynomial.
    5. $\mathcal{V}$: outputs 1 if all checks pass; otherwise outputs 0.
```

Fiat-Shamir Transformation: The Eval protocol in FRI PCS is a public coin interactive argument of knowledge. In the online phase, $\mathcal{V}$ sends only random elements as challenges. Using the standard Fiat-Shamir transformation, this interactive protocol can be made non-interactive. Specifically, $\mathcal{V}$'s random challenges are derived by hashing all prior messages.

Sub-Proofs in FRI PCS: The FRI PCS involves three main types of sub-proofs:

- **Merkle Tree Openings:** Prove that received entries have correct indices. Verify consistency between the reconstructed and received Merkle tree roots.

- **Algebraic Equation Verification:** Verify the correctness of algebraic equations over the opened entries. Note: This includes TapSTARK verifications beyond TapFRI.
- **Fiat-Shamir Validation:** Ensure that hash outputs of previous prover messages match the used challenges.

3. Building Blocks

We introduce several building blocks used in TapSTARK.

3.1 Bitcoin Script

Bitcoin employs a scripting system to govern transactions. A Bitcoin script is essentially a list of instructions attached to each transaction, specifying the conditions under which the recipient can spend the transferred Bitcoins. Although not Turing complete, the Bitcoin script supports a range of operations through Opcodes. These include checking string equality or inequality, performing basic arithmetic operations (e.g., addition), and computing hash values for given inputs. Each integer or string processed by the script is limited to a maximum of 32 bytes. The scripting language also supports conditional logic, such as ‘if-else‘ constructs.

3.2 Bit Commitment

A commitment scheme is a cryptographic primitive that enables one to commit to a value while keeping it hidden, with the ability to reveal it later. Such schemes have two key security properties: **hiding** and **binding**. Hiding ensures the commitment does not disclose any information about the committed value, while binding guarantees that the committed value cannot be opened as two distinct values.

The bit commitment used in BitVM is a weaker form of commitment designed for bits within the Bitcoin script. Specifically, the bit commitment for a bit b is represented as $C_b = \{\mathsf{H}(\boldsymbol{x_0^b}), 0 \,||\, \mathsf{H}(\boldsymbol{x_1^b}), 1\}$, where $\boldsymbol{x_0^b}$ and $\boldsymbol{x_1^b}$ are secret strings. To open C_b, the prover provides b and $\boldsymbol{x}$. The script performs: Unlocking C_b as 0 if $\boldsymbol{x} = \boldsymbol{x_0^b}$ and $b = 0$; Unlocking C_b as 1 if $\boldsymbol{x} = \boldsymbol{x_1^b}$ and $b = 1$; Otherwise, it does not unlock.

The bit commitment is **hiding** due to the non-invertibility of hash functions. However, it lacks cryptographic binding, as C_b can technically be opened as both 0 and 1. Instead, it relies on a weaker binding property enforced by a fraud-proof mechanism: if the prover opens C_b as both 0 and 1, a challenger can detect this and claim a BTC reward. To uphold this weak binding, the secret strings (hash pre-images) for any two commitments—or within the same commitment—must be distinct.

3.3 Taptree

Taptree (defined in BIP-341) is a Merkle-style hash tree introduced by Greg Maxwell, Peter Wuille, and Andrew Poelstra. It enhances the privacy, efficiency, and flexibility of Bitcoin's scripting capabilities, enabling complex script designs with minimal on-chain data.

In a Taptree, each leaf node represents a Bitcoin script, while the root (referred to as Taproot) acts as the identifier for the unspent transaction output (UTXO). To unlock the UTXO, a user must provide: 1) a valid input satisfying one of the leaf scripts, and 2) a valid Merkle path corresponding to the script.

This structure reduces on-chain storage requirements by omitting temporarily unused scripts, enhancing scalability and privacy.

3.4 Federation

In TapSTARK, only the Taproot, partial scripts, and the relevant Merkle tree paths are eventually stored on-chain. This approach poses a challenge: participants cannot easily verify that the entire Taptree is correctly constructed.

To overcome this, TapSTARK introduces a Federation mechanism. After the prover generates a proof and the corresponding Taproot, the Federation verifies the construction of the complete Taptree. This includes validating all Bitcoin scripts and ensuring the correctness of the Merkle tree structure.

4. TapSTARK

Now, we introduce the TapSTARK principle to address the aforementioned sub-proofs.

4.1 Representation and Commitment of Field Elements

TapFRI utilizes bit commitments to represent and commit field elements. To commit to a field element $a \in \mathbb{F}_p$ with $n = \log_2 p$, the prover decomposes a into bits $(a_1, \dots, a_n)$ (assuming little-endian order without loss of generality). The commitment to a is constructed as a Bitcoin script comprising $C_{a_1}, \dots, C_{a_n}$, where each bit commitment C_{a_i} is defined as:

$$C_{a_i} = \{\mathsf{H}(\boldsymbol{x_0^{a_i}}), 0 || \mathsf{H}(\boldsymbol{x_1^{a_i}}), 1\}.$$

Here, for any i, j, $b \neq b'$, and $b, b' \in \{0, 1\}$, the following conditions hold:

$$\boldsymbol{x_b^{a_i}} \neq \boldsymbol{x_{b'}^{a_i}}, \quad \boldsymbol{x_b^{a_i}} \neq \boldsymbol{x_b^{a_j}}, \quad \boldsymbol{x_b^{a_i}} \neq \boldsymbol{x_{b'}^{a_j}}.$$

To open a from $\mathsf{Com}(a)$, the prover supplies the inputs $(\boldsymbol{x_{a_1}^{a_1}}, a_1, \dots, \boldsymbol{x_{a_n}^{a_n}}, a_n, a)$. The verifier runs the Bitcoin script with these inputs, which:

1. Unlocks the bit commitments to reveal $a_1, \dots, a_n$.

2. Computes $a' = \sum_{i=1}^{n} a_i \cdot 2^{i-1}$.
3. Unlocks if and only if all bit commitments are valid and $a' = a$.

These operations for generating a from $a_1, \ldots, a_n$ are embedded directly in the Bitcoin script for the commitment to a. While Bitcoin scripts do not natively support multiplication operations, they simulate multiplication through repeated addition.

4.2 Construction and Opening of Taptrees

The bit commitment suffices to commit to and open a single field element. However, for verifying an entry within a Taptree, it is necessary to validate both the Bitcoin script and the associated Taptree paths. Unlike the programmable verifier in STARK, the Taptree path validity in TapSTARK is fixed and executed by Bitcoin miners. These miners, as determined by Bitcoin's consensus, only validate the given Taptree paths and do not ensure that the paths correspond to the entries specified by the Fiat-Shamir transformation. This limitation allows a malicious prover to cheat by selectively opening preferred entries in the Taptree without detection.

To address this issue, an additional commitment to each entry's index is incorporated into every Tapleaf of the Taptree. For instance, if the target field element resides in the i-th entry of the Taptree, the Bitcoin script for this Tapleaf includes a commitment to i.

The index i is bit-decomposed as $(i_1, \ldots, i_m)$. Similar to C_a, the commitment C_i is composed of $C_{i_1}, \ldots, C_{i_m}$. To unlock the script, the prover provides the inputs:

$$\boldsymbol{x}_{\boldsymbol{i_1}}^{\boldsymbol{i_1}}, i_1, \ldots, \boldsymbol{x}_{\boldsymbol{i_m}}^{\boldsymbol{i_m}}, i_m, i; \quad \boldsymbol{x}_{\boldsymbol{a_1}}^{\boldsymbol{a_1}}, a_1, \ldots, \boldsymbol{x}_{\boldsymbol{a_n}}^{\boldsymbol{a_n}}, a_n, a.$$

The script unlocks if and only if all the following conditions are satisfied:

1. All bit commitment scripts for i and a unlock successfully.
2. $i = \sum_{j=1}^{m} i_j \cdot 2^{j-1}$.
3. $a = \sum_{j=1}^{n} a_j \cdot 2^{j-1}$.

4.3 Verification of Algebraic Equation Consistency

After opening entries on Taptree leaves, the consistency of algebraic equations involving these entries must be verified. For instance, in the first round of Algorithm 1, to ensure that

$$(\beta, f_0(\beta)), \quad (-\beta, -f_0(\beta)), \quad (\alpha_1, f_1(\beta^2))$$

lie on a common degree-1 polynomial, the verifier equivalently checks the algebraic equation:

$$f_1(\beta^2) = \frac{f_0(\beta) + f_0(-\beta)}{2} + \alpha_1 \cdot \frac{f_0(\beta) - f_0(-\beta)}{2}, \tag{1}$$

where $f_0(\beta)$ and $f_0(-\beta)$ are derived from the Taptree commitment to $f|_L$, and $f_1(\beta^2)$ is obtained from the Taptree commitment to $f_1|_L$.

The Bitcoin script performing this check takes $f_0(\beta), f_0(-\beta), f_1(\beta^2), \alpha_1$ as inputs, stores bit commitments for these values, and implements the consistency verification based on Equation (1). The script unlocks only if all bit commitments are valid and the equation holds true.

4.4 Validation of Value Consistency

An important requirement in the above checks is ensuring the consistency of values reused across multiple steps. For instance:

1. The bit commitment to $f_0(\beta)$ is employed in both Taptree opening and algebraic equation verification.
2. A single Bitcoin script may not encapsulate an entire check due to size constraints, necessitating multiple scripts that reference the same value.

This consistency is inherently guaranteed by the properties of bit commitments. The unique secret strings within bit commitments ensure that commitments for the same or different values are distinct. Additionally, the validity of Bitcoin scripts—including whether they reference the same bit commitments—is verified by the Federation.

4.5 Verification of Fiat-Shamir Transformation Validity

In TapSTARK, challenges for interactive $(\lambda, \alpha_1, \ldots, \alpha_{\mathsf{r}+1})$ and query $(\beta_1, \ldots, \beta_q)$ phases are generated using the Fiat-Shamir transformation.

Bitcoin scripts for this validation take as inputs:

- The previous Taproot of the Taptree commitment for an intermediate polynomial.
- The corresponding hash output.

These scripts verify that the hash of the Taproot matches the provided hash input, ensuring the correctness of the Fiat-Shamir transformation.

5. Comparison

ZKPs	Transparency	BTC Script Size	Need OP-CAT
Groth16	No	1 GB	No
FFlonk	No	0.95 GB	No
Circle Stark	Yes	4 MB	Yes
TapSTARK (this work)	Yes	15.7 MB	No

Table 1: ZKP Implementations Comparison

Recently, several other ZKPs have been implemented on Bitcoin. We compare them with our TapSTARK here to highlight the contributions of our scheme. Table 1 summarizes these implementations across key dimensions (Transparency, Bitcoin Script Size, and OP-CAT), emphasizing the advantages of our approach. These dimensions primarily reflect the current execution environment of the Bitcoin network.

The first dimension is **transparency**, meaning no trusted setup is required. Transparency is crucial for building a decentralized system and aligns with Bitcoin's vision. The second dimension is **script size**, which is constrained by Bitcoin's consensus. Currently, a Bitcoin transaction is limited to 400 KB, though it can be increased to 4 MB through cooperation with a Bitcoin miner. A larger script size in a ZKP verifier results in more Taproot leaves and more bit commitments, thereby increasing the cost. The final dimension is whether the ZKP verifier relies on **OP-CAT**, a Bitcoin proposal whose approval status remains uncertain.

Groth16 and FFlonk, implemented in the BitVM project and improved by the community, suffer from large script sizes and lack transparency. These issues are avoided in TapSTARK. While Circle Stark is more efficient than other implementations, it relies on a non-activated proposal.

6. Conclusion

We introduces TapSTARK, a Bitcoin-adapted ZKP protocol based on the STARK framework. It integrates advanced cryptography with Bitcoin's ecosystem to enable efficient on-chain proof verification while maintaining compatibility with Bitcoin's scripting constraints. Key technologies include: Field elements are committed using compact bit commitments implemented as Bitcoin scripts. Replacing traditional Merkle trees with Bitcoin's native Taptree, TapSTARK optimizes storage and leverages Bitcoin's consensus for verifying paths. Algebraic equations and reused values are verified using bit commitment properties and a Federation mechanism to prevent inconsistencies. Challenges for proof phases are generated and validated through Fiat-Shamir transformations within Bitcoin scripts. Therefore, TapSTARK avoids trusted setups and non-standard Bitcoin proposals like OP-CAT, offering a transparent and scalable ZKP solution tailored for Bitcoin's limitations.

References

1. Linus R, Aumayr L, Zamyatin A, et al. BitVM2: Bridging Bitcoin to Second Layers[J]. presented by ZeroSync, TU Wien, BOB, University of Pisa, University of Camerino, and Common Prefix, 2024.

2. Ben-Sasson E, Bentov I, Horesh Y, et al. Fast reed-solomon interactive oracle proofs of proximity[C]//45th international colloquium on automata, lan-

guages, and programming (icalp 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

3. Groth J. On the size of pairing-based non-interactive arguments[C]//Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35. Springer Berlin Heidelberg, 2016: 305-326.

4. Gabizon A, Williamson Z J. fflonK: a Fast-Fourier inspired verifier efficient version of PlonK[J]. Cryptology ePrint Archive, 2021.

5. Haböck U, Levit D, Papini S. Circle STARKs[J]. Cryptology ePrint Archive, 2024.