# Charms: Programmable Assets on Bitcoin and Beyond

**Ivan Mikushin**[1,2] **Gadi Guy**[2] **Andrew Throuvalas**[2]

[1]Charms Inc

[2]BitcoinOS

May 2025

## Abstract

Bitcoin remains the heart of the crypto economy, yet its poor programmability and scalability have capped its potential. Enter Charms — a revolutionary protocol that "enchants" Bitcoin, enabling programmable and portable assets natively on its ledger. These assets are chain-agnositc, and can be seamlessly "beamed" to various other UTXO blockchains, allowing them to live natively on any integrated chain. Leveraging client-side validation of recursive zkVM proofs, Charms eliminates the need for bridges, trusted validators, or transaction graph traversal. The result: truly decentralized, client-side validated smart assets that inherit Bitcoin's security while transcending its limitations. Inspired by Bitcoin's pioneering metaprotocols and Cardano's eUTXO model, Charms is a tesseract representing the first post-chain standard able to integrate builders and users with the existing dapp ecosystems of more scalable chains beyond Bitcoin. This is the end of wrapped tokens and the beginning of Bitcoin's full integration into the programmable Web3 economy.

# 1 Introduction

Bitcoin (1) emerged as a beacon of freedom - the first peer-to-peer digital money system that could liberate humanity from the control of central banks and financial intermediaries. The network's stability and decentralization have allowed it to prosper as "digital gold," growing into a $2T juggernaut at the heart of the crypto economy.

However, the very permanence that makes Bitcoin valuable has also been its curse. Its inflexible design and cultural resistance to change left it unable to deliver on its full revolutionary potential. Its limitations in scalability, privacy, and programmability drove developers to spawn an entire parallel ecosystem of competing chains and tokens seeking greater decentralized functionality. But in doing so, these projects cut themselves off from Bitcoin's unmatched security, repute, network effects, and massive capital base. The result is a fragmented crypto landscape divided between Bitcoin's rock-solid foundation and an innovative but siloed world of decentralized applications - each unable to benefit from the other's strengths.

Thankfully, several years of advancements in both UTXO technology and zero-knowledge cryptography have assembled the pieces to an entirely new paradigm for augmenting the motherchain, while melting its borders between the wider Web2 and Web3 worlds.

That paradigm is called Charms — a programmable, portable, and developer-friendly app protocol for Bitcoin. Charms opens Bitcoin to all manner of crypto innovation, including fungible tokens, NFTs, lending protocols, DEXes, and more — all powered by Bitcoin and without sacrificing decentralization.

This protocol enables the first digital assets that simultaneously satisfy three important qualities:

- They have the same direct ownership as Bitcoin (can be sent to a Bitcoin address as a Bitcoin transaction output).
- They are programmable in ways that appeal to mainstream developers — using general purpose languages, like TypeScript or Rust.
- They can be transferred across chains and used in any ecosystem while maintaining their smart contract logic, without needing wrapped tokens or fragile bridges.

Charms are "unchained" because their state is verified by recursive zk-proofs, not in any single ledger, so they can materialize natively on whatever chain hosts the proof — no locks, no bridges, no custodians. Charms transcend blockchains as we know them today. This makes them not just a theoretical boon for the Bitcoin ecosystem, but a highly practical pathway to bootstrapping an advanced economy of Bitcoin tokens, which can leverage successful infrastructure already present on other chains. One of those chains is Cardano — the first key technology that informed the creation of Charms.

# 2 Precursors

## 2.1 Cardano

Cardano demonstrated that the UTXO model is, in fact, programmable – and has been for a long time.

Cardano pioneered the eUTXO (Extended UTXO) model (2, 3), in which

- Multiple assets exist in a transaction output,
- the output also holds a piece of arbitrary data (called datum), and

- the output can be spent by a transaction that satisfies a predicate (encoded in Plutus Script, a Cardano native bytecode level language) by presenting another piece of data (called redeemer).

We discuss improvements over Cardano's smart-contract design below (in **ToAD — an Extended Extended UTXO Model**).

### 2.1.1 Extended UTXO Model

The Cardano blockchain pioneered the approach: the eUTXO model allows storing in transaction outputs not just ADA (the system native token in Cardano), but also user created Cardano native tokens (called so because the Cardano ledger natively enforces token preservation, just like for ADA, without invoking smart-contracts). Cardano transaction outputs can also contain datum — a piece of arbitrary data, effectively a smart-contract state associated with the output. So, a Cardano UTXO can contain:

- an amount of ADA (required)
- any amounts of any tokens (optional)
- datum (optional)

Spending scripts (smart contracts) have the following signature:

```
fn spend(
    datum: Option<Data>,
    redeemer: Data,
    utxo: OutputReference,
    tx: Transaction,
) -> bool
```

For a transaction to be able to spend outputs locked in such script, Cardano nodes will have to run the script for each such output.

If the script is implemented naively, a transaction trying to trade multiple outputs, each for the same amount of Ada, might result in a single output mistaken as the payment for each individual traded item — this is known as the double satisfaction problem (4, 5). The solution is to look at the whole transaction (all spent and created outputs), effectively running the exact same computation as many times as there are outputs spent from the smart contract.

There are other script types in Cardano (6), which don't look at individual outputs and run only once per transaction (if needed, e.g. if an asset is minted, the asset's mint script is run).

## 2.2 Ordinals and Runes

Upon popularizing in early 2023, Bitcoin metaprotocols like Ordinals and Runes reimagined how we can build on Bitcoin. They demonstrated that:

- Arbitrary size data can be put into a Bitcoin transaction,
- Digital assets can be created fully on Bitcoin and sent over to a Bitcoin address
- Client side validation is a practical way to extend Bitcoin without changing Bitcoin itself

Charms would not exist without Ordinal or Runes. However, they also wouldn't exist if these standards were already programmable. Without programmability, advanced token applications such as Bitcoin IDs and yield bearing stablecoins remain out of reach.

## 2.3 zkVM Technology

zkVMs (Zero Knowledge Virtual Machines) enable developers to write provable programs in general purpose programming languages (like Rust) and generate succinct (zkSNARK) proofs of their execution. This allows for verifiable computation without revealing the underlying data, enhancing privacy and scalability in blockchain applications.

Prior to zkVMs (effectively, until 2024 when zkVM projects started releasing to production), ZK proofs could only be generated by running specially formatted input data (data items would have to be elements of a large finite scalar field) through so called ZK circuits — specially constructed programs with a single fixed length execution path (for example, branching logic has to be emulated).

zkVMs were the missing piece of infrastructure that made Charms much easier to build, allowing Charms apps to be written in Rust (vs ZK circuits) and proven within tolerable time (seconds to minutes — depending on the app complexity) and verified in milliseconds.

## 2.4 ToAD — an Extended Extended UTXO Model

Charms started as ToAD (Tokens as App Data) (7) — a fresh take on an Extended UTXO model.

ToAD was proposed as an improved ledger model for zkBitcoin (8). A transaction involving zkBitcoin apps ("zkapps") would have to satisfy a validation predicate with a signature

$$F : (\text{ins}, \text{outs}, x, w) \to \text{Bool}$$

where:

- ins — set of outputs spent by the transaction,
- outs — set of outputs created by the transaction,
- $x$ — public *redeeming* (or *spending*) data necessary to validate the transaction (a great example would be a set of *spending signatures*).
- $w$ — private *witness* data necessary to validate the transaction (e.g. pre-images of hashes in the public data).

Each zkapp UTXO should have

- a map of: **validation predicate → state data**
  - e.g. **token policy → amount**:
    - $T_1 \to a_1$
    - $T_2 \to a_2$
    - $T_3 \to a_3$
  - e.g. **smart-contract validator → smart-contract data**
    - $S_1 \to d_1$
    - $S_2 \to d_2$

If a zkBitcoin transaction spends or creates any number of zkapp UTXOs, then **all** validation predicates used in those UTXOs need to be satisfied to validate a transaction.

ToAD was an attempt to introduce a token model, (1) enjoying the benefits and (2) avoiding the problems of Cardano eUTXO, and (3) do it on Bitcoin.

Charms applies minor changes to ToAD to level up the eUTXO model: from extended UTXO to enchanted UTXO.


# 3 Techniques

## 3.1 Charms UTXO Model

The ToAD model, with minor changes, is adopted by Charms. Charms *app contracts* have the following shape:

$$F : (\text{app}, \text{tx}, x, w) \to \text{Bool}$$

In Charms,

- app contracts use tx (the whole transaction, listing ins and outs),
  - ins, outs are lists of UTXOs
  - UTXOs are
    - identified by UTXO ID (256-bit byte string + integer index), and
    - contain *charms*: map App → Data, where each individual entry `app → data` is called a charm.
- app itself is an argument to the app contract — it has
  - the `vk` — verification key,
  - `identity` — useful when the same contract can power multiple assets / apps, and
  - `tag` — a single character (with two values having special meaning, but free to be used otherwise).

Data must be an unsigned integer (`u64`) for fungible token assets (apps with `tag == TOKEN`), arbitrary data otherwise.

Outputs (UTXOs) can be created (by transactions) and then spent (by other transactions) — these are the only two states of Charms outputs, just like in the underlying UTXO ledger.

Now, because outputs can contain multiple charms bound together (we call them *strings of charms*), and they are passed to app contracts as such, it allows for composability of apps. For example, an order-book exchange app data would be a limit order, specifying the quote token and the minimum price for a base token in the same output.

Charms UTXO model achieves all design goals of ToAD **and** it can potentially add support for any underlying UTXO-based ledger, relying solely on the client and the underlying blockchain (without need to trust anyone). This ability to move across chains is powered by:

- the abstract nature of the Charms ledger model (designed to be a layer on top of a UTXO ledger),
- recursive zkVM proofs.

## 3.2 Recursive zkVM Proofs

Charms client library doesn't need to traverse the transaction history. All it needs to make sure a transaction has correct Charms metadata (we call such metadata a *spell*) is a succinct (Groth16 (9)) zkVM proof.

Spell proof in a transaction attests to the following statements being true:

(i) All pre-requisite transactions indeed produced the charms in their outputs: their spell proofs are correct.
(ii) All Charms app contracts in this transaction are satisfied: their proofs are correct.

Therefore, if we're looking at a Bitcoin transaction included in a block (which happens to be a part of the main chain) with a correct Charms spell, we know two facts about it:

(i) It is spending and creating legitimate outputs — ensured by Bitcoin consensus.
(ii) Charms spent and created by the transaction are legitimate — as attested by the zkVM proof.

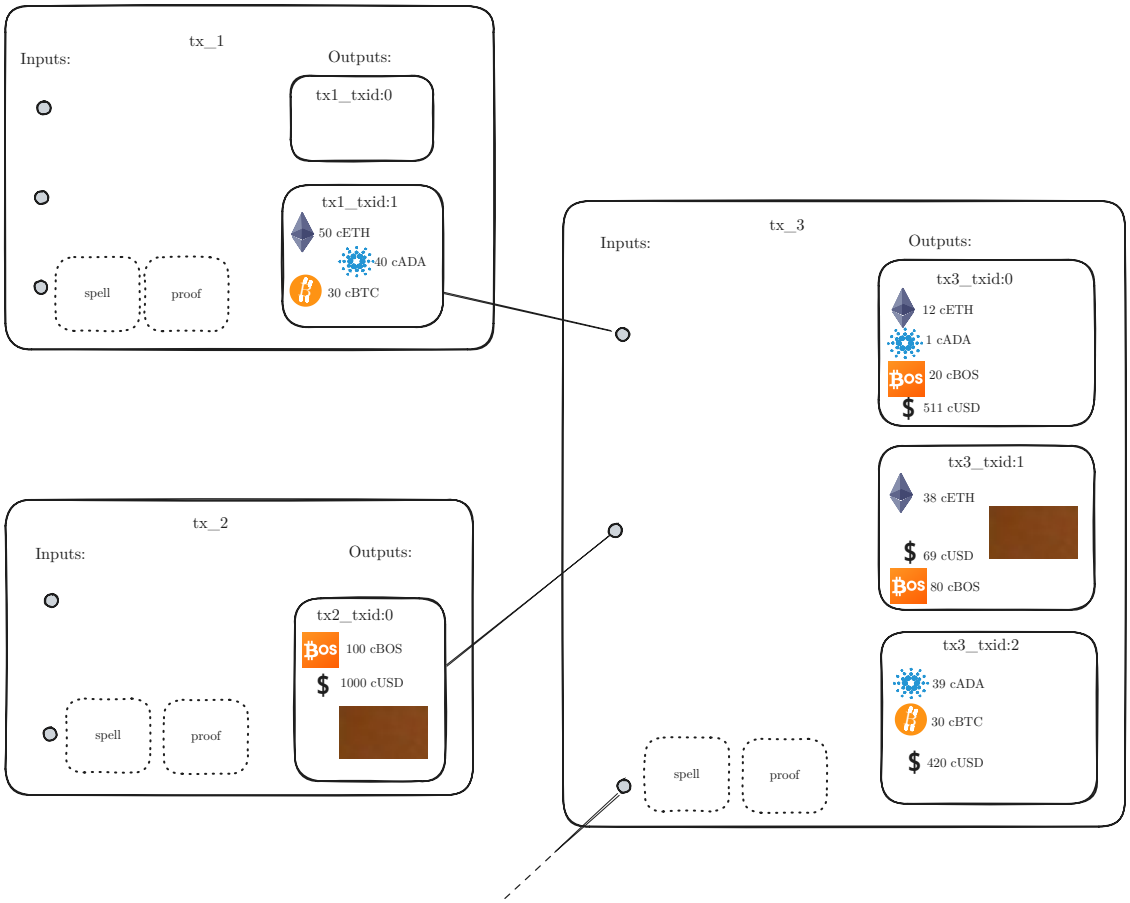
So, all it takes to know that Charms in transaction outputs are legitimate, is read the spell and verify the associated Groth16 proof: all the necessary data is available in the transaction (and nowhere else).

This makes every client a Charms validator, even if it is a web or a mobile app.

## 3.3 Client Side Validation

This is precisely what every Charms client is doing. The underlying blockchain requirements still apply, so the client needs to talk to a Bitcoin node to be able to get transactions creating the client's Bitcoin outputs. To read what Charms are in these outputs, the client reads the spells and verifies proofs (in the transactions creating those outputs).

# 4 Charms Data Model




## 4.1 App Contracts

Charms exist to make programmable assets possible on Bitcoin. Put simply, Charms **are** programmable tokens on top of Bitcoin UTXOs.

Programmability exists to do one thing: enable *apps*. Charms apps can implement:

- fungible and non-fungible tokens,
- DEXes, auctions and lending protocols,
- … you (create and) name it.

App state needs to be stored somehow, and that's what charms (as tokens) are for.

A single *charm* is a token, NFT or, generally, a fragment of app state pertaining to a particular output. Structurally, it is an entry of a mapping `app -> data` on top of a Bitcoin UTXO. You can have as many as you want such entries, creating a *string of charms* (a Charms output).

Combining tokens, NFTs and arbitrary apps in *strings of charms* allows for **composability**:

- a limit order to trade one token for another,
- artist royalty policies for NFTs
- bridging
- … limitless other things.

A *string of charms* gets created or spent as one unit, just like a Bitcoin UTXO. *Charms* can **only** exist on top of UTXOs (such outputs are said to be *enchanted*). Because of this, whoever owns the Bitcoin UTXO, can do whatever they want with *charms* in it (even destroy them).

Fungible tokens and NFTs are treated as special cases:

- a fungible token data (the value of a charm) is its amount (a positive integer, e.g. `69420`),
- NFTs carry **arbitrary** data.

Charms Apps are essentially predicates that Charms transactions must satisfy:

```
pub fn app_contract(app: &App, tx: &Transaction, x: &Data, w: &Data) -> bool
```

In the above signature,

- `app: &App` refers to the app itself, so that the predicate could find the app's own data in the transaction,
- `tx: &Transaction` is the transaction for which the predicate is being evaluated,
- `x: &Data` — app-specific (on-chain) public input,
- `w: &Data` — app-specific (off-chain) private input.

We detail the datatypes below.

### 4.1.1 App

```
pub struct App {
    pub tag: char,
    pub identity: B32,
    pub vk: B32,
}
```

`App` is a tuple of `tag/identity/VK` (see `App`) where:

- `tag` is a single character representing the app type (with special values: `n` for NFTs, `t` for fungible tokens). `tag` can be anything (with `n` and `t` treated specially: simple transfers of NFTs and fungible tokens don't need app contract proofs).
- `identity` — a 32-byte array uniquely identifying the asset (among others with the same tag and implementation).
- `VK` — a 32-byte array representing the verification key hash of the app implementing the logic of the asset: how it can be minted or burned, staked, etc.

### 4.1.2 Data

Data represents any CBOR-serializable data value (must implement `Serialize` and `Deserialize`). `charms-data` library provides convenient API to convert to and from the app-specific datatype.

### 4.1.3 Transaction

```
pub struct Transaction {
    /// Input UTXOs.
    pub ins: BTreeMap<UtxoId, Charms>,
    /// Output charms.
    pub outs: Vec<Charms>,
}
```

In UTXO model (e.g. as in Bitcoin, Dogecoin, Cardano), transactions spend inputs (previously created UTXOs) and create outputs (new UTXOs).

Charms transactions are no different: spend inputs, create outputs.

### 4.1.4 Charms

```
pub type Charms = BTreeMap<App, Data>;
```

The `Charms` type represents the content of a Charms output or a *string of charms* — multiple charms bound together (in a single UTXO). This is simply a mapping `App -> Data`. A single entry of this mapping is called a *charm*, and it represents one asset of potentially many in a single output.

In case of fungible tokens, `Data` encodes a positive integer (`u64`) — the amount of the fungible token (specified by `App`) in the Charms output. It can be anything for other types of assets.

## 4.2 Spells and Proofs

Spells are the magic that creates *charms*.

The idea is to add some metadata to Bitcoin transactions that would tell the client software that a transaction deals with Charms (inspired by Runes' *runestones* and Ordinals' *inscriptions*). We call this metadata a *spell* because it "magically" *enchants* the transaction and creates *charms*.

*Spells* are **client-side validated**, meaning that **the clients** choose to interpret or ignore them.

A *spell* is said to be *correct* if and only if all of these are true:

- it is successfully parsed and interpreted
- makes sense for the transaction (e.g., doesn't produce more Charms outputs than there are Bitcoin outputs)
- has a valid proof

*Correct spells* can create, destroy and transfer *charms* (programmable assets). *Incorrect spells* are ignored.

Double-spending is prevented by Bitcoin, so the spell's proof only needs to prove that the spell itself is correct. We go one step further and guarantee, that given the transaction is valid and the spell is correct, it's sufficient to verify the proof to ensure the spell is correct, i.e. there is **no need to traverse transaction history** to make sure the transaction is dealing with legitimate assets.

### 4.2.1 On-Chain Binary Representation of a Spell

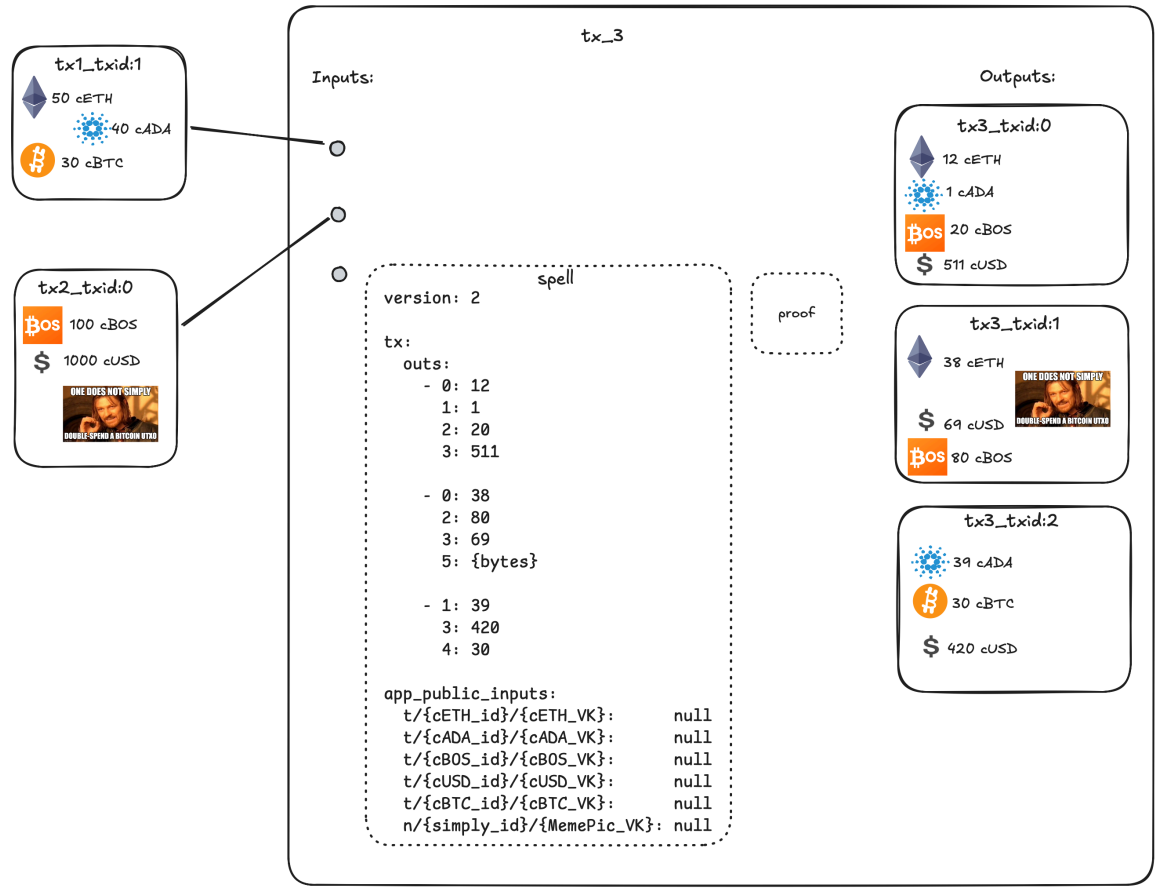
A spell is included in a Taproot witness of the underlying Bitcoin transaction, in an *envelope* — a sequence of opcodes `OP_FALSE OP_IF … (push data) … OP_ENDIF`, which is effectively a no-op: since the condition is `false`, no data is pushed onto the stack.

```
OP_FALSE
OP_IF
  OP_PUSH "spell"
  OP_PUSH $spell_data
  OP_PUSH $proof_data
OP_ENDIF
```

where:

- `OP_PUSH "spell"` shows that the envelope contains a **spell**.
- `OP_PUSH $spell_data` — CBOR-encoded `NormalizedSpell`.
- `OP_PUSH $proof_data` — Groth16 proof attesting to verification of correctness of the spell.

### 4.2.2 Structure of a Spell




A spell consists of:

- the protocol version
- the transaction (see `NormalizedTransaction` below)
  - inputs,
  - outputs
- public inputs for all involved apps

### 4.2.3 `NormalizedSpell`

```
pub struct NormalizedSpell {
    pub version: u32,
    pub tx: NormalizedTransaction,
    pub app_public_inputs: BTreeMap<App, Data>,
}
```

`NormalizedSpell` encodes the spell structure in normalized form (to minimize size). The involved Apps are enumerated in `app_public_inputs` (with potentially empty `Data`). Since it is a sorted map, we can use the App's integer index to refer to the App within the spell.

### 4.2.4 `NormalizedTransaction`

```
pub struct NormalizedTransaction {
    pub ins: Option<Vec<UtxoId>>,
    pub outs: Vec<NormalizedCharms>,
```

```
    pub beamed_outs: Option<BTreeMap<u32, B32>>,
}
```

`NormalizedTransaction` is the normalized encoding of a Transaction (to minimize size).

`ins` field is optional because a spell is a metadata of a Bitcoin transaction: we can easily recover the list of input UTXO IDs from the underlying transaction.

`outs` is a list of Charms outputs, where each is encoded as `NormalizedCharms`.

`beamed_outs` is discussed in the **Beaming Charms** section below.

#### 4.2.5 `NormalizedCharms`

```
pub type NormalizedCharms = BTreeMap<usize, Data>;
```

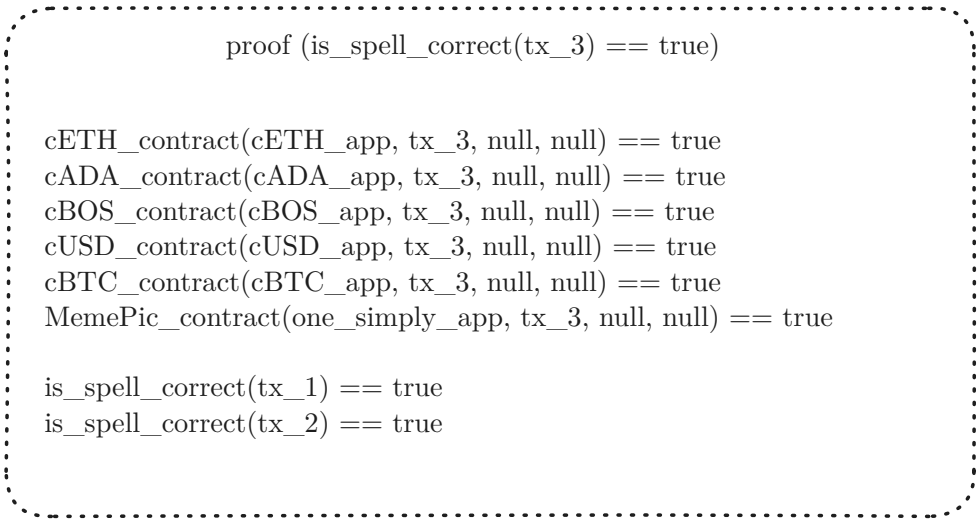
`NormalizedCharms` represents a string of charms within a `NormalizedSpell` with Apps replaced by integer indexes (in `NormalizedSpell.app_public_inputs`).

#### 4.2.6 Proof

Spell proof is recursive: it's a Groth16 proof of successful run of a zkVM program that verifies:

- the spell is well-formed
- the transaction satisfies all involved app contracts
- all spells from pre-requisite transactions (i.e. those creating the UTXOs we're spending) are correct

## 5 Recursive zkVM Proofs




As we mentioned, spell proofs are recursive: a correct spell proof attests to

- verifying the spell is well-formed
- verifying app contract proofs
- verifying pre-requisite spell proofs ← this is the recursive part

This removes the need to traverse transaction history to confirm that the charms spent by a transaction were indeed there. So, with a single Groth16 proof verification, we know if charms in transaction outputs are good.

## 5.1 Spell is Well-formed

The following must hold for a spell to be well-formed.

- Protocol version is the one current one supported by the software.
- Apps from **all** input and output charms are listed (with their public inputs) and **only** those apps are listed.
- There are no indexes in normalized charms outside the list of apps (higher than `number_of_apps - 1`).

## 5.2 App Contract Proofs

An app contract proof is a zkVM proof that the app contract is satisfied by the transaction. The app (compiled to a RISC-V binary) is run by the zkVM, and the proof that it runs successfully is generated, with the following public and private inputs.

Public inputs:

- `App`
- `Transaction`
- `x: Data` — additional app-specific public input

Private inputs:

- `w: Data` — app-specific private input

These proofs are then verified against the app verification key (`app.vk`) when constructing spell proofs.

There are two special cases that don't require presenting app proofs:

- simple transfer of fungible tokens,
- simple transfer of NFTs.

Fungible tokens are implemented by Apps with tag `'t'` (`app.tag == TOKEN`). NFTs are implemented by Apps with tag `'n'` (`app.tag == NFT`).

Simple transfer of a fungible token means, the total amount of the fungible token in input charms and output charms of a transaction is exactly the same.

Simple transfer of an NFT means that the NFT stays exactly the same in an output as it was in an input of the transaction.

## 5.3 Spell Proofs

Charms software includes a program `charms-spell-checker` compiled to RISC-V binary and run in a zkVM. The proof that it runs successfully is generated and then wrapped into a Groth16 proof, with the following public and private inputs.

Public inputs:

- Spell VK — the recursive spell verification key. It is used to verify pre-requisite spell proofs.
- NormalizedSpell — the spell being checked in normalized form.

Private inputs:

- app contract proofs — proofs that app contracts are satisfied by this spell.
- pre-requisite transactions — transactions that created Charms outputs spent or read by this spell

App contract proofs are verified against corresponding app VKs.

Spells and their proofs from pre-requisite transactions are extracted and verified against the spell VK.

### 5.4 Protocol Upgradability

Spells have `version` field, informing which spell VK to use when verifying the spell proof. Known versions and corresponding spell VKs are known constants (within the Charms software which is open source).

When a prerequisite transaction has a spell with a known protocol version, it is parsed accordingly and its proof is verified against the spell VK for that version.

This enables simple and transparent upgradability of the protocol.

## 6 Beaming Charms

The meta-protocol design decoupled from the underlying blockchain empowers Charms to move to other chains (and back to Bitcoin). We call this capability *beaming* as it reminds of transport beaming depicted in Star Trek, where people or objects would disappear completely from one place (say, on some planet) and fully materialize in a different place (say, the transporter room of starship Enterprise).

### 6.1 Datatype Support

`NormalizedTransaction` has an optional field **`beamed_outs`** to mark Charms outputs (*strings of Charms*) as *beamed* to other chains. The keys are indices of the beamed outputs (in this `NormalizedTransaction`), and the values are SHA256 hashes of destination UTXO IDs on the target blockchains.

```
pub struct NormalizedTransaction {
    pub ins: Option<Vec<UtxoId>>,
    pub outs: Vec<NormalizedCharms>,

    /// Mapping from the output index to the destination  UTXO ID hash.
    pub beamed_outs: Option<BTreeMap<u32, B32>>,
}
```

This signals that charms are no longer assigned to the output. They cannot be “unlocked”, because they are not here anymore.

#### 6.1.1 Protocol

The protocol is executing a transfer of assets from chain B (e.g. Bitcoin) to chain C (e.g. Cardano).

(i) **on chain C**: create a “placeholder” UTXO (or the destination UTXO)
- transactions creating such outputs don’t carry any spells
- do not spend this UTXOs yet!

(ii) **on chain B**: create a Charms output that is being “beamed” to chain **C** — added to the mapping in **`beamed_outs`** with the hash of the “placeholder” UTXO ID on destination chain **C** (created in step 0).

After these steps, the Charms output can be considered moved to **chain C** and is good to be spent there.

(iii) A spell spending such output **on chain C** is correct if and only if it has a proof that:

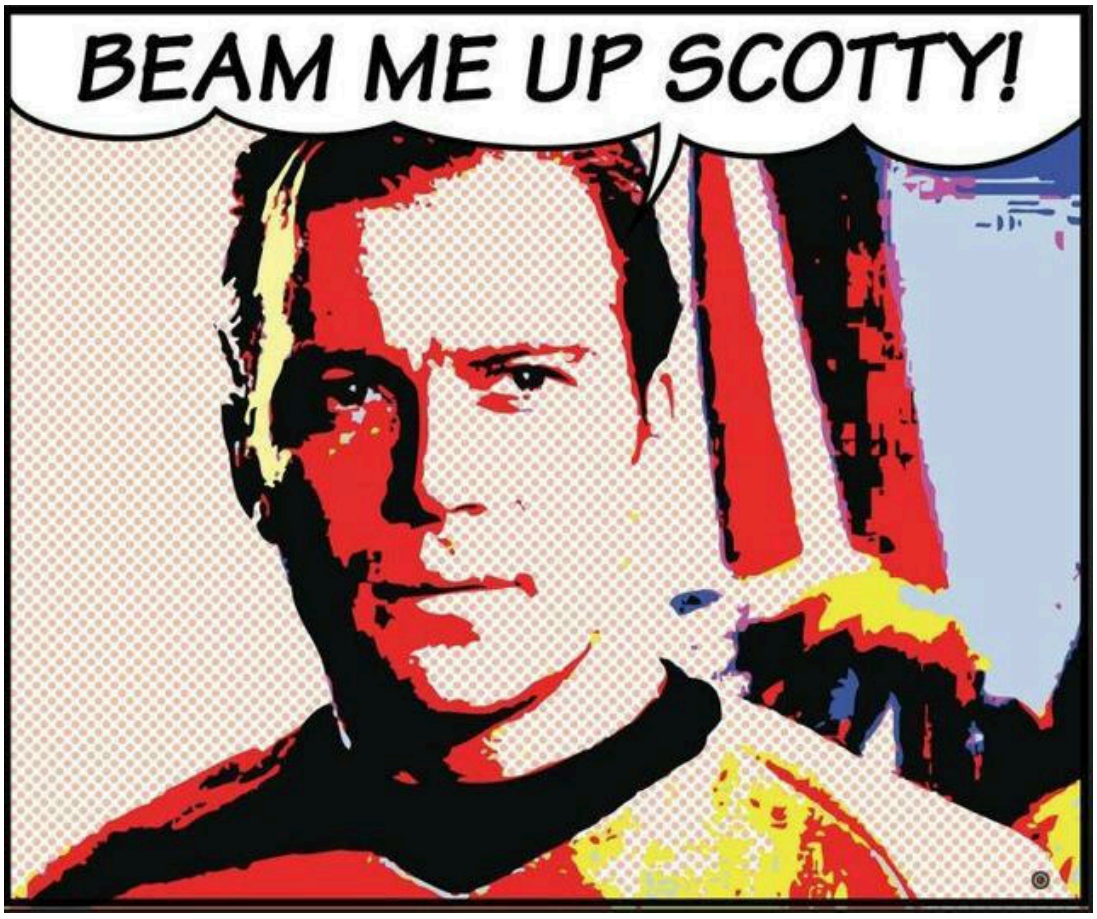
- a transaction **on chain C** has created the beam destination output (this is taken care of by the underlying blockchain),
- the spell would be correct without the beaming if the charms were put in the beam destination UTXO directly **on chain C**,
- a correct spell **on chain B** (with valid proof) created the output with a corresponding entry in **`beamed_outs`** with the hash of the destination UTXO ID on chain C,
- the beaming transaction (on chain B) is included in the main branch of the blockchain history.

To prove transaction inclusion in the main branch on Bitcoin, we can provide

- Merkle proof of transaction inclusion in a block,
- proof of sufficient work performed on top of the block (to mine subsequent blocks).

We can visualize this process as:

(i) a transaction on the destination chain (**C**) create "placeholder" for Charms to be sent to,
(ii) a transaction on the source chain (**B**) *beams* the Charms into their "placeholder" (on the destination chain, **C**).




### 6.1.2 Effect on Apps

App contracts **don't care about chains**: there are no mentions of any chains (neither **B**, nor **C**) in app contract code. `UtxoId` struct (which is used in app contracts) has the same format regardless of the chain.

Thus apps **become cross-chain without trying** to become cross-chain.

### 6.1.3 Additional Considerations

If **multiple** strings of charms are sent to a single UTXO on the target chain, only **one** of them can be spent. The spending transaction decides which one. The other ones are effectively destroyed.

# 7 How it Works Put Together

The full cycle of building a Charms app (e.g. a Bitcoin backed stablecoin) looks something like this:

(i) Run `charms app new shiny-new-token` in your shell, and get a fully buildable Rust project that can be customized to whatever it needs to do.
(ii) Build the app contract functionality. The only limit is the signature of the entry point function, the app contract predicate. Standard Rust can be used, no compromises. Use `charms app build` to compile the app to the zkVM-ready RISC-V binary.
(iii) Test the app contracts with `charms app run` against different potential spells you anticipate.
(iv) Build front-end and back-end of the app and interact with Charms' Rust or web APIs (or the CLI) to generate spell proofs and embed them into Bitcoin transactions to be signed by users' wallets.
(v) Profit!
(vi) Go cross-chain without changing a line of code in app contracts.
(vii) More profit.

Operation flow for Charms apps (which also hints at why Charms app implementations are called **app contracts**) is the following:

(i) Create your transaction / spell. It can do **anything**, as long as is **satisfies** the **app contracts** for all involved charms.
(ii) Generate the zkVM proof that the spell is correct using Charms library, API or CLI. In order to do this, provide previous transactions, plus proofs of inclusion for those on different chains. The result is a transaction (enhanced with the spell and proof, or dare we say, *enchanted*) ready to sign and submit to the blockchain.
(iii) Sign and submit the transaction.

The above flow is very similar to interaction with UTXO blockchains in general, with the only difference that validation happens off-chain and is attested to by a zkVM proof.

# 8 What's Next

## 8.1 Wasm App Contracts

Currently, Charms apps are written in Rust and compiled to RISC-V ELF binaries — to be run and generate proofs by a zkVM.

RISC-V ELF is a popular choice of CPU architecture and binary format among zkVM vendors, which means that currently only Rust can practically be used to implement Charms app contracts.

We will continue using Rust to build Charms (it is fantastic as a systems programming language), but we also want to lower the barrier for app developers, and therefore will look into using Wasm (WebAssembly) as the binary format for Charms apps.

We will work with zkVM vendors to achieve this.

## 8.2 Charms as Native Tokens on Smart UTXO Chains

If the destination chain has smart contracts (token policies and locking scripts) that can verify SNARK proofs over a pairing-friendly elliptic curve, such as BLS12-381 (which Cardano does), it is possible for Charms tokens (both fungible and non-fungible) exist on that chain as native tokens.

We will briefly describe how this would work on Cardano.

### 8.2.1 Charms as Cardano Native Tokens (CNTs)

On the high level, the end goal is:

- when Charms tokens are transferred to Cardano (in a wallet supporting both Bitcoin and Cardano), they show up on Cardano as CNTs:
  - when beamed to Cardano, tokens are minted as CNTs
  - when beamed from Cardano, they are burned
- when Cardano tokens are transferred to Bitcoin (or other Charms-enabled chains), they are treated in a similar fashion:
  - tokens are locked on the Cardano side and minted as Charms
  - when transferred back, they are burned as Charms and unlocked on Cardano

The idea is to use a proxy Cardano token policy for Charms tokens. This token policy will allow minting and burning tokens based on verification of a ZK proof of the transaction in question satisfying the Charms tokens' app contracts.

We will also introduce a proxy spending script for Charms apps to act as Cardano smart contracts. It will effectively introduce portable zkVM based smart contracts (implemented in Rust) to Cardano.

We also need a proxy Charms app for original Cardano tokens coming to Charms. Minting and burning of original CNTs is only possible when they are operated natively on Cardano. Aside from minting and burning, original CNTs are going to be fully operational as Charms tokens on any Charms-enabled blockchains.

## 8.3 More Chains

More than $80bn total market cap is in UTXO-based blockchains other than Bitcoin, including the largest ones, Dogecoin ( $34bn), Cardano ($27bn). About half (the smallest chain being larger than $6bn in market cap) allow storing sufficient additional data in transactions (as evidenced by existence of their respective versions of Ordinals) to be able to carry a Charms spell (and proof). We are looking to build Charms support for these blockchains.

# 9 Use Cases

The revolutionary capabilities of Charms enable entirely new categories of blockchain applications. Here we explore three transformative use cases that showcase the protocol's potential to reshape both Bitcoin functionality and broader Web3 interoperability.

## 9.1 Unchained Bitcoin (xBTC)

We can wrap BTC into a Charms token - xBTC - introducing two groundbreaking upgrades to the world's largest digital asset.

First, xBTC brings programmability to BTC, much like how wETH enabled smart contract functionality for ETH. This allows BTC holders to participate in advanced DeFi applications

and smart contract interactions that were previously impossible with native BTC - directly upon the Bitcoin ledger.

Second, xBTC eliminates the need for traditional bridging infrastructure to move BTC between chains. Once BTC is locked and xBTC is minted, the token can move freely between any Charms-integrated chains without requiring additional locking or minting operations. This dramatically reduces friction compared to conventional bridging solutions that require separate infrastructure for each chain pair. Finally, this opens avenues for less trusted methods of moving BTC between chains - particularly for chains that cannot verify external events, including Litecoin and Dogecoin.

The result is a truly portable BTC that maintains its security while gaining unprecedented functionality and cross-chain mobility.

### 9.2 Decentralized Bitcoin Onramp

Charms could revolutionize bitcoin acquisition by creating a decentralized, peer-to-peer, censorship-resistant onramp that sidesteps traditional KYC requirements.

This system would enable users to purchase xBTC (a wrapped BTC implemented as a Charm) directly paying with, say, CashApp without any intermediary's knowledge. When a buyer sends cash via CashApp, the system leverages zero-knowledge proofs to cryptographically verify the transaction occurred without revealing personal details.

Once verified, a Cardano smart contract automatically releases the equivalent xBTC from sellers' escrowed funds to the buyer. The platform processes what appears to be regular peer-to-peer transfers, so CashApp isn't aware it is processing a P2P Bitcoin purchase. This creates a truly private, non-KYC pathway to Bitcoin ownership that embodies core cypherpunk principles of financial privacy and freedom from surveillance.

### 9.3 Self-Auditing Stablecoin

Charms' ability to verify off-chain information through zero-knowledge proofs enables an entirely new category of transparent, verifiable stablecoins. Consider a stablecoin that can only mint new tokens when it has cryptographically verified sufficient backing in a traditional financial account.

For example, a stablecoin could be programmed to verify the balance of a custodian's CashApp account before allowing any new minting. While this doesn't eliminate custodial risk entirely, it provides:

- Real-time verification of backing assets
- Transparent proof of reserves
- Automated compliance with backing requirements
- Reduced trust assumptions around custody

This creates a new standard for stablecoin transparency and verification, where backing can be continuously monitored and proven rather than relying solely on periodic attestations.

These examples represent just a fraction of what's possible with Charms. As the protocol matures and more developers begin building with it, we expect to see an explosion of innovative applications that leverage Charms' unique capabilities to create more secure, interoperable, and powerful blockchain solutions.

## 10 Conclusion

We have presented Charms — a programmable asset protocol for enchanting Bitcoin alongside other UTXO-based ledgers (like Cardano).

By using recursive zkVM proofs, client-side validation, and a novel Extended UTXO model, Charms straps Bitcoin with an unchained and secure metalayer for building the decentralized internet of value. Charms doesn't just give Bitcoin apps — it makes Bitcoin omniscient.

This is made possible by advancements in zkVM technology that powers our recursive ZK proofs, eliminating the need to traverse blockchain transaction history. This allows clients to be very thin and removes the need for indexers (for the purpose of validating transactions) or any infrastructure other than the underlying blockchain.

By bootstrapping Charms on Cardano's dApp ecosystem, we empower developers to build freely — with Cardano projects becoming the first to explore this new programmable frontier. As the rest of the crypto industry collapses inward toward Bitcoin's gravity, Cardano becomes the launchpad.

Our short-term development goals are to migrate to Wasm as the binary format for app contracts, opening up to the broader app/web2 developer community, finish implementing Charms integration with Cardano native tokens and begin expansion to other UTXO chains (like Litecoin and Dogecoin).

Charms is the culmination of years of evolution in UTXO-based smart contracts. It's not a bridge, nor a Layer 2, but a new paradigm — the unchained token standard.

## 11 Acknowledgements

We would like to thank Edan Yago, Rainer Koirikivi, Ricart Juncadella, James Aman and the rest of BitcoinOS team for insightful discussions and support that helped envision and realize Charms and for reviewing this white paper.

Succinct is a remarkable group of people who have built SP1, a developer-friendly open source zkVM, which we enjoyed using when building Charms.

We are also grateful to Maksym Petkus for reviewing this paper and for questions helping to sharpen our thinking, as well as David Wong (of zkSecurity) who has done most of the work behind zkBitcoin, the project Charms grew from.

## References

1. NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System. Online. 2008. Available from: https://bitcoin.org/bitcoin.pdf

2. MANUEL M. T. CHAKRAVARTY, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, Joachim Zahnentferner, James Chapman. UTXOma:UTXO with Multi-Asset Support. Online. 2020. Available from: https://iohk.io/en/research/library/papers/utxomautxo-with-multi-asset-support/

3. MANUEL M. T. CHAKRAVARTY, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, James Chapman and WADLER, Philip. The Extended UTXO Model. Online. 2020. Available from: https://iohk.io/en/research/library/papers/the-extended-utxo-model/

4. Plutus Reference. Common Weaknesses: Double Satisfaction. Online. 2022. Available from: https://github.com/cardano2vn/plutus/blob/main/doc/reference/common-weaknesses/double-satisfaction.rst

5. Aiken Documentation. Fundamentals: Common Design Patterns. Online. 2024. Available from: https://aiken-lang.org/fundamentals/common-design-patterns

6. Aiken Documentation. Fundamentals: EUTXO Crash Course. Online. 2024. Available from: https://aiken-lang.org/fundamentals/eutxo

7. IVAN MIKUSHIN, David Wong. zkBIP-001: Tokens as App Data (ToAD 🐸). Online. 2024. Available from: https://github.com/CharmsDev/zkbitcoin/blob/main/zkBIPs/zkBIP-001.md

8. DAVID WONG, Ivan Mikushin. zkBitcoin: Zero-Knowledge Applications for Bitcoin. Online. 2024. Available from: https://github.com/CharmsDev/zkbitcoin/blob/main/whitepaper.pdf

9. GROTH, Jens. On the size of pairing-based non-interactive arguments. 2016.