Moria Protocol: A Decentralized Stablecoin and Loan System on Bitcoin Cash

Dagur Valberg Johannsson

2024

Abstract

The Moria Protocol is a decentralized, censorship-resistant borrowing protocol on the Bitcoin Cash (BCH) network, allowing users to borrow stablecoins backed by BCH collateral. Designed to be capital-efficient and secure, Moria enables users to access loans while maintaining minimal collateral requirements. The protocol leverages the Bitcoin Cash UTXO model and is tailored to operate within its consensus limitations. This whitepaper provides an in-depth analysis of Moria's system architecture, loan processes, collateral management, and potential future developments.

1 Introduction

1.1 Motivation

The explosive growth of decentralized finance (DeFi) has demonstrated high demand for stablecoin-backed loans, allowing users to leverage assets without sacrificing exposure to their primary holdings. While DeFi lending protocols have become commonplace on other blockchains, Bitcoin Cash (BCH) — known for its scalability and low transaction fees — has not had a native, decentralized solution for loan issuance and stablecoin minting.

The Moria Protocol represents a first-of-its-kind innovation for the Bitcoin Cash network, made possible by the CashTokens upgrade in May 2023[6], which introduced new capabilities for smart contract functionality on BCH. Leveraging Bitcoin Cash's UTXO model, Moria is designed to operate efficiently within BCH's consensus rules. By enabling users to collateralize BCH for stablecoin minting, Moria expands the utility of Bitcoin Cash as a DeFi platform, supporting the growth of decentralized financial applications. The first loan on mainnet was initiated in a transaction at block `866303`[12], marking a milestone in the protocol's deployment.

While the Moria Protocol is tailored for Bitcoin Cash, similar collateral-backed stablecoin protocols exist on other platforms, such as the Ethereum-based Liquity protocol, which has achieved significant adoption with a total value locked of over 333 million USD[11].

1.2 Bitcoin Cash Preliminaries

Bitcoin Cash employs the Unspent Transaction Output (UTXO) model, with a notable feature allowing UTXOs to store both tokens and BCH. Moria leverages this structure for simplicity and efficiency: the Moria contract itself encodes an NFT to maintain its local state, the loan contract stores each loan's state information in its own NFT, and the fungible token represents the core asset. This arrangement allows collateral (in BCH) and stateful NFTs (representing loan or protocol data) to coexist within a single UTXO.

Additionally, Bitcoin Cash's scripting language supports introspection, an essential feature for Moria's design, enabling it to enforce contract constraints and securely manage interactions between BCH, tokenized assets, and contract state.

2 Definitions

Borrower The entity that initiated a loan by providing collateral in exchange for asset tokens. A borrower owns a loan as long as minimal collateral requirements are held.

Liquidator The entity that repays a defaulted loan, redeeming the collateral.

Loan Default A situation where a loan contract fails to meet the minimum collateral requirement. This situation allows for the loan to be liquidated.

Collateral Assets pledged by the borrower in Bitcoin Cash (BCH), to secure the loan.

Oracle A contract that provides up-to-date asset price information, used by the Moria protocol to determine loan-to-collateral ratios.

NFT Non-Fungible Token are a token type in which individual units cannot be merged or divided, each NFT contains a commitment[6], used by the Moria protocol to store stateful data including loan ownership and oracle price sequence.

P2PKH Is the most common script used for locking an output to someone's public key.[5]

UTXO Unspent Transaction Output. Represents the remaining balance from previous transactions that can be used as input in new transactions.

PKH Public Key Hash. A HASH160 of a public key. Used together with a cryptographic signature to prove ownership of a Loan contract.

Moria Treasury The Moria asset tokens held by the Moria contract UTXO that have not been lent out to borrowers.

Oracle A contract or service that provides external data, such as asset prices, to the blockchain. In the Moria protocol, the oracle ensures accurate and up-to-date pricing information for maintaining loan collateral ratios.

Asset An external asset that the Moria token represents and is the asset priced in the Oracle.

3 System Architecture

3.1 Incentives

The Moria Protocol relies on economic incentives to maintain stability in the loan and stablecoin ecosystem. By leveraging natural market forces, Moria encourages borrowers and asset holders to interact in ways that stabilize the protocol and keep collateral levels balanced. These incentives are particularly crucial in periods of price volatility, whether in bull or bear markets.

3.1.1 Borrower Incentives

Token Undervaluation: If the value of the borrowed token decreases significantly, existing borrowers are incentivized to repurchase the token from the market at a reduced price to repay their loans. This opportunity allows them to settle their debt for less than the original loan value, effectively benefiting from the token's undervaluation.
Token Overvaluation: When the borrowed token price is high, new borrowers are encouraged to take out loans to profit from the token's overvaluation.

3.1.2 Market Dynamics

However, these incentives may have limitations under extreme market conditions:
Bull Markets: In bull markets, speculators might be willing to borrow even at "below-market" collateral ratios, using loans to leverage their exposure to appreciating assets. This behavior can lead to increased demand for loans, potentially affecting the collateral requirements and overall stability of the system.
Bear Markets: In bear markets, borrowers may prioritize unlocking their BCH collateral, competing with others who view BCH as a risky asset to use as collateral. This competition could result in heightened demand for loan repayments, as participants seek to secure their positions in BCH amid declining asset values.

3.1.3 Incentivized Floating Peg

The borrowed tokens in the Moria Protocol operate with an "incentivized floating peg." Built-in market incentives encourage the borrowed token's value to remain close to the underlying collateral asset's value, allowing the token to

float while still gravitating towards parity. This incentivized structure is designed to promote price stability through borrower actions, ensuring the token remains usable as a stable unit without requiring direct intervention.

3.2 Overview of Components

The Moria Protocol consists of three smart contracts designed to work together on the Bitcoin Cash network: the Moria contract, the Loan contract, and the Oracle contract. These contracts interact to enable decentralized, collateralized loans and stablecoin issuance. The contracts are written in CashScript, a high-level smart contract language for Bitcoin Cash[10].

3.2.1 Moria Contract

The Moria contract acts as the central treasury for the protocol, managing the issuance and repayment of stablecoin assets and enforcing collateral requirements. When a user requests a loan, the Moria contract verifies that the collateral meets the minimum required ratio (110%) based on asset prices provided by the Oracle contract. The contract operates under a 0% interest model, allowing users to access loans without incurring interest charges.

3.2.2 Loan Contract

The Loan contract manages individual loan positions and collateral holdings. Each loan has a unique UTXO, which includes an NFT that encodes the loan information, the loan ownership (borrower) and amount of asset loaned. Borrowers can add additional collateral to their loan if needed, helping prevent liquidation in the event of price fluctuations. When a loan falls below the required collateral threshold, the Loan contract allows other users to repay the loan and claim the collateral.

3.2.3 Oracle Contract

The Oracle contract provides up-to-date price data for the Moria Protocol, ensuring that collateral values remain accurate in relation to the market. Implemented as an on-chain contract, the Oracle encodes current asset prices within its NFT commitment. The overcollaterization incentices users to keep track of the collaterized loans as they can profit from liquidating bad loans, protecting the protocol from undercollateralized loans.

3.3 Interactions

These three contracts interact to create a secure, decentralized lending system:

- **Loan Issuance**: When a loan is requested, the Moria contract checks collateral value against the Oracle's price data and creates a corresponding Loan contract to hold the collateral. This process enforces the minimum

collateral ratio (110%) and prevents loans from being issued without sufficient collateral.

- **Collateral Management**: Borrowers can add collateral to their loan contract to avoid liquidation if the asset price declines, thus maintaining capital efficiency while reducing default risks.
- **Repayment and Liquidation**: If the collateral meets the minimum requirements, only the borrower may repay the loan and retrieve their collateral. However, if the collateral value falls below the threshold, the loan is open for liquidation, allowing others to repay the loan and claim the collateral.

This structure enables Moria to function as a decentralized stablecoin and loan protocol on Bitcoin Cash, emphasizing capital efficiency and security within the network's UTXO model.

4 Contract Functionalities

4.1 Oracle contract

Moria uses an on-chain oracle where the current asset price is encoded in its NFT as defined in table 1. With the exception of the `oracle owner` field, the encoding is the same as defined for General Protocol oracles[3].

Field	Data type	Size	Description
Oracle owner	Bytes	20 bytes	PKH of Oracle owner.
Message Timestamp	LE signed Integer	4 bytes	Unix timestamp in UTC and seconds for the moment the oracle produced this message.
Message Sequence	LE signed Integer	4 bytes	Sequence number for this price message relative to all of this oracle's messages.
Data Sequence	LE signed Integer	4 bytes	Sequence number for this price message relative to all of this oracle's price messages.
Data Content	LE signed Integer	4 bytes	Price of the asset.

Table 1: NFT commitment of an oracle contract

The interface to the oracle has the following requirement:

- The Oracle MUST accept being used at any input index, and be re-created at the same output index.
- The Oracle MUST not require a fee of more than 1000 satoshis.
- The Oracle MUST accept the argument '1' as its first and only input, encoded as a VM number, indicating that it's used for its price information.

With Moria, we have used the d3lphi oracle[2] developed by the same author as the Moria protocol. Further implementation details of the oracle contract are not discussed in this paper.

4.2 Moria Contract

The Moria contract holds the treasury of the deployed Moria protocol. Users interact with the contract when creating new loans and repaying existing loans.

A loan allows users to take fungible tokens out of the contract treasury, while a repayment allows them to return them.

Moria contract follows the asset price using an oracle contract. The sequence number of the oracle is stored in the contracts commitment and enforcing a forward sequence ensures that only new oracle price messages can be used.

4.2.1 NFT encoding

The Moria contract encodes the last sequence number of the Oracle contract. The sequence number is to enforce that old oracle messages are not used. The sequence number can only be incremented. Encoding is shown in Table 2.

Field	**Data type**	**Size**
Oracle sequence	VM Number [1]	Variable

Table 2: NFT commitment of Moria

4.3 Deployment

A valid Moria deployment is a transaction that creates a new token category with a fixed number of fungable tokens and a NFT that encodes the current oracle contract with 'minting' capabilities, where both the fungable tokens and the NFT is sent to the contracts locking script in a single UTXO. The redeem script of Moria must encode the token category of the oracle contract and the locking script of the loan contract.

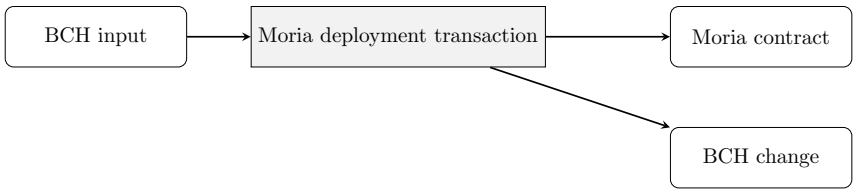

Figure 1: Transaction that deployes Moria

The fixed number of fungible tokens sets the ceiling on how much can be lend out by the protocol. Figure 1 shows a transaction deploying a new contract.

- The Moria contract SHOULD create its own category when deployed to ensure that no tokens or NFTs exist outside the system.
- The Moria contract SHOULD NOT encode an old oracle sequence number, as this would allow borrowers to use old price information when borrowing assets.

- The Moria contract NFT MUST have 'minting' capabilities, as it will need this capability to create loan NFTs.
- The Moria contract must not be deployed to multiple UTXOs with the same token category. Creating multiple UTXOs would create a race condition, where the oracle sequence number may differ, introducing an attack vector where oracle prices could differ.

4.3.1 Updating the oracle sequence

The Moria contract stores in its NFT commitment the last Oracle sequence number it knows of. Only oracle messages with the same sequence number or later can be used as input for price information.

An oracle contract may have many UTXO threads where not all are up-to-date. To ensure the Moria contract uses updated price information, the update sequence call allows to updating the internally stored sequence number.

The update sequence call is also used when repaying a loan (see section 4.4.4).

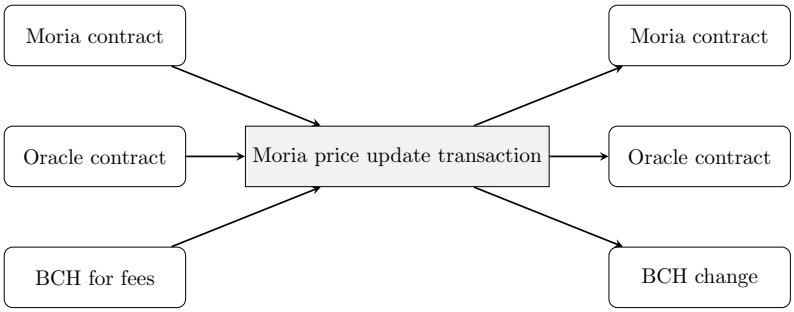

Figure 2: Transaction updating the Moria asset price

A update sequence transaction takes the Moria and Oracle contracts as inputs and re-creates them at the same output indexes as visualized in Figure 2.

4.3.2 Borrowing Assets

Borrowing is the process of taking asset tokens from the Moria contract treasury, giving them to the borrower and having the borrower locking collateral into a Loan contract in exchange.

The borrowed asset cannot exceed the minimal collateral requirement (110%). This is enforced by the contract. The calculation of the collateral requirement is calculated using price information from the Oracle contract.

Additionally, the Moria contract enforces that the Loan contract encodes into its NFT commitment the PKH of the borrower and the token amount lent.

A transaction borrowing asset must take both the Moria and Oracle contract as input, re-create them as outputs and create a Loan UTXO containing the collateral. This is visualized in Figure 3.

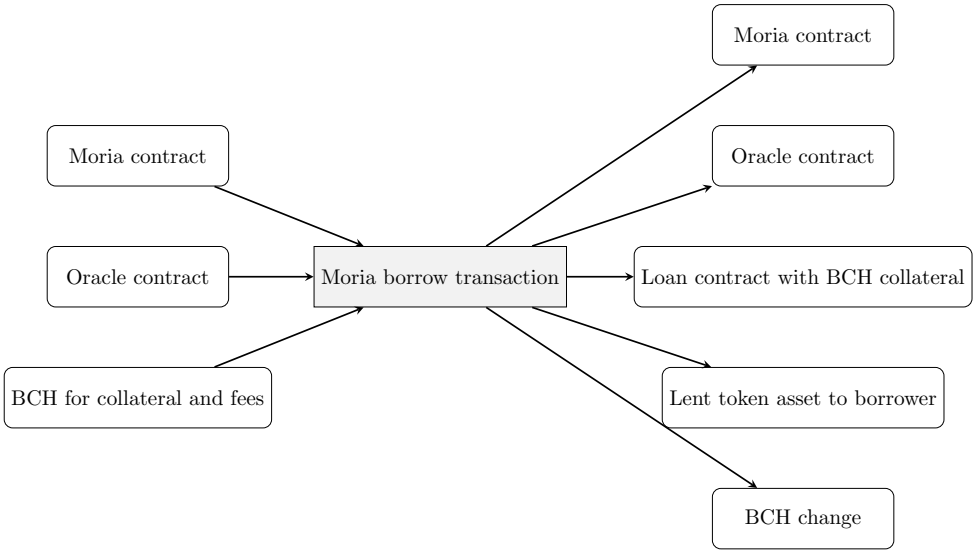

Figure 3: Transaction that borrows a token asset

4.4 Loan Contract

A loan contract UTXO is created for each individual loan, holding the collateral provided by the borrower. This UTXO remains under the borrower's control as long as the collateral value meets or exceeds the minimum collateral ratio of 110%. Should the collateral fall below this threshold, the loan is marked as "in default," indicating a risk to the Moria protocol's stability.

In the case of a default, the collateral is open to redemption by any participant. The first party to repay the outstanding loan balance can claim the collateral, profiting from the difference between the repayment cost and the collateral's value.

Each loan UTXO also includes an NFT commitment, which encodes the specific details of the loan, such as the collateral amount, borrower information, and loan terms.

4.4.1 NFT encoding

The Loan contract encodes the owner of the loan and collateral. It also encodes the number of asset tokens loaned. Encoding is shown in Table 3.

The NFT does not have minting or mutable capabilities.

Field	Data type	Size
Loan owner PKH	Bytes	20 bytes
Borrowed token amount	VM Number[1]	Variable

Table 3: NFT commitment of a loan utxo

4.4.2 Deployment

A loan contract is deployed by the Moria contract when lending occurs. See section 4.3.2.

The NFT must not have minting or mutable capabilities. This is enforced by the Moria contract.

4.4.3 Adding additional collateral

The owner of the loan has permission to add collateral. Adding additional collateral is a feature to avoid liquidation when the price movement of Bitcoin Cash is trending downwards.

There is a minimum requirement of adding 100000 satoshis additional collateral. This limit is to mitigate an attack where a malicious borrower is flooding the utxo to prevent liquidation by external parties.

4

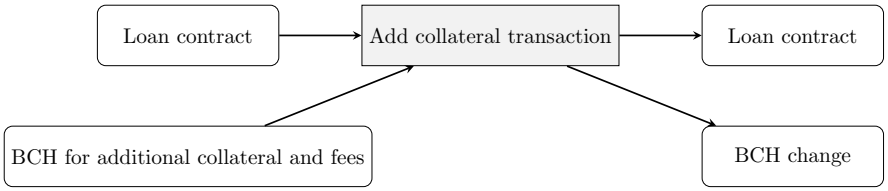

Figure 4: Transaction that adds additional collateral to a loan

4.4.4 Repay loan

The `repayLoan` method is used to redeem collateral. This requires full repayment of the asset token borrows. These asset tokens are payed back into the Moria contract.

The borrower can redeem the collateral. Additionally, if the loan has defaulted, anyone can redeem the collateral.

The oracle contract input is used to determine if a loan has defaulted.

4.4.5 Redeem by borrower

If the loan fulfills the minimum collateral requirement (110%) then only the borrower can redeem the loan. In this case, he has to provide a valid public key and signature, as he would when unlocking a P2PKH UTXO.

4.4.6 Redeem by liquidator

If the loan has defaulted then a dummy public key and signature can be provided as input instead of the borrowers public key and signature (a 33 byte and 64 byte blob of data that will be ignored representing public key and signature).

4.4.7 Transaction construction

Figure 4 shows a transaction structure for adding collateral. The moria contract is used as input to repay it the token assets. The loan is used as input to reclaim collateral, note that this UTXO is not re-created. The oracle contract is used as input to determine if the loan has defaulted.

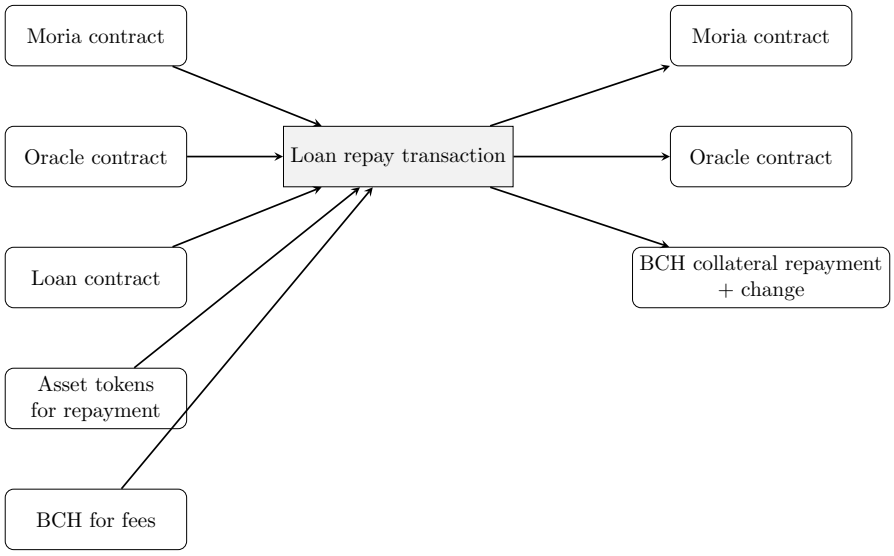

Figure 5: Transaction that redeem collateral

4.5 Security and Risk Mitigation

4.5.1 Collateral

A high collateral ratio is essential for asset stability. Delays in liquidating defaulted loans could lead to depreciation of the token asset. By enforcing a minimum collateral threshold, the protocol provides a buffer against sudden price drops, reducing the impact of unaddressed defaults.

In a case of a undercollaterized loan, it would be reasonable to expect the market to devalue the token asset until it becomes sufficiently low to incentivize liquidation of undercollaterized loans.

4.5.2 Liquidation Process

An efficient liquidation process is integral to managing undercollateralized loans within the protocol. When a loan falls below the required collateral ratio, the protocol permits any participant to repay the outstanding loan balance and claim the associated collateral. This mechanism not only provides an incentive for users to monitor the system for defaulted loans but also mitigates the accumulation of bad debt. By promptly liquidating undercollateralized positions, the protocol can minimize the impact of defaults on overall asset stability.

4.5.3 Oracle Trustworthiness

Since the Bitcoin Cash network does not have native access to external data, the Moria Protocol relies on an on-chain oracle to supply accurate price information. This dependency introduces potential risks, as incorrect or compromised data may affect collateral ratios and loan stability.

A key mitigation strategy involves using an aggregated oracle, where multiple independent entities provide price data to enhance reliability and reduce the impact of any single point of failure. The d3lphi Oracle[2] supports this approach, allowing for decentralized data publication from multiple sources to maintain accurate and resilient price feeds.

4.5.4 UTXO Flooding

The Moria protocol operates in a "single-threaded" manner, meaning it exists at all times within a single UTXO. A potential attack vector involves a malicious actor repeatedly interacting with the protocol's UTXO (covenant), effectively blocking borrowers from accessing it for loan issuance or repayment.

To mitigate this risk, the Moria protocol enforces a minimum cost for contract interactions. In addition to the network fee, the Oracle contract imposes a fee of 1000satoshis, loan repayments require a minimum repayment amount, and collateral additions must meet a minimum threshold of 100000 satoshis. These fees make such an attack costly, as UTXO flooding would require the contract to be used multiple times every second.

4.6 Miner Extracted Value (MEV)

The risk of Miner Extracted Value (MEV) arises when miners selectively exclude certain transactions to exploit opportunities, such as liquidations. This is an inherent network limitation within Bitcoin Cash. For example, a miner might exclude "add collateral" transactions to push a loan into default, allowing them or others to liquidate the loan.

Historically, MEV has not been a significant issue on Bitcoin Cash, likely due to the limited incentives available for such exploitation. In cases where malicious miners are present, borrowers can mitigate potential impacts by adding collateral or repaying loans promptly, thereby increasing the likelihood that honest miners will include their transactions.

Research has been conducted on protocol-level solutions to mitigate MEV risks. Techniques such as the Subchains approach, as outlined in [13], propose additional incentives for miners to behave honestly, potentially reducing the impact of MEV in networks like Bitcoin Cash.

5 Testing

The Moria protocol has a set of functional system tests[8] that run on a local blockchain simulating Bitcoin Cash network called "regtest". This is the same local network that the leading Bitcoin Cash consensus node uses for their on functional tests [7]. These tests cover all calls to Moria and Loan contract.

6 Future Development

This section discusses future improvements for the Moria Protocol.

6.1 Simplify existing contract

The current contract is size optimized to fit within strict rules of execution cost within the Bitcoin Cash consensus rules. The Bitcoin Cash network will perform an upgrade in May 2025[4] that reduce restrictions for smart contracts. This allows for reducing the complexity of the Loan contract by separating repayment and liquidation into two distinct features as well as separating sequence update and token repayment on the Moria contract.

6.2 NFT representing loan ownership

As implemented, a loan ownership is secured using a public key hash in effectively the same way funds are secured in a normal Pay-to-Public-Key-Hash (P2PKH) contract. This method can be replaced securing the loan with a HASH256 of a token category and commitment, allowing a NFT to own the loan. The loan contract can use transaction introspection to see that this NFT has been provided as an input to the transaction, proving ownership.

Using NFT to enforce a loan allows for transferring the loan by transferring the NFT. Using NFT will also allow a loan contract to be embedded into other smart contracts.

6.3 Fee based on recent loan activity

With the Bitcoin Cash May 2025 upgrade, there will be room for contract code that weights recent loan activity. This will allow the contract to detect excessive loan activity and enforce a fee for new loans.

An Exponentially Weighted Moving Averages (EWMA) weighting is a good candidate for this. Each time a new loan is made, update the EWA based on the time difference since the last update. This allows the weight to decay over

time without needing to store past values. The Oracle contract already includes a timestamp can be used and there is room in the Moria commitment to include a WEMA weight and an earlier timestamp.

7 Acknowledgments

The development of the Moria Protocol benefited from the insights of participants in the discussion thread on Bitcoin Cash Research, where this contract was initially posted[9]. The input from the Bitcoin Cash community has been valuable in shaping the protocol's design, and the author thank everyone who contributed to the discussion on the forum and in private.

8 Conclusion

The Moria Protocol introduces a decentralized stablecoin and loan system tailored for the Bitcoin Cash network. By leveraging Bitcoin Cash's UTXO model, CashTokens capabilities, and on-chain oracles, Moria enables efficient, secure collateralized loans and stablecoin issuance without centralized control.

Through a combination of economic incentives and mitigation strategies, Moria supports a stable token ecosystem adaptable to market fluctuations. It offers a scalable solution for collateral-backed lending on Bitcoin Cash, aligning with the growing demand for DeFi applications on the network. Future enhancements to Bitcoin Cash's smart contract functionality may further streamline Moria's contract design, paving the way for additional features and expanded use cases. As a pioneering protocol on Bitcoin Cash, Moria lays the groundwork for a broader ecosystem of decentralized finance applications, fostering innovation in censorship-resistant financial infrastructure.

References

[1] VM number as implemented in CashScript. Available at: https://libauth.org/functions/vmNumberToBigInt.html.

[2] d3lphi Oracle Contract. Available at: https://gitlab.com/riftenlabs/moria/oracle-contract.

[3] General Protocol Oracles Specification. Available at: https://gitlab.com/GeneralProtocols/priceoracle/specification.

[4] CHIP-2021-05: VM Limits – Targeted Virtual Machine Limits. Available at: https://github.com/bitjson/bch-vm-limits/tree/master.

[5] Bitcoin Cash Specification. "Transaction Locking Scripts in Bitcoin Cash." Available at: https://reference.cash/protocol/blockchain/transaction/locking-script.

[6] Token Primitives for Bitcoin Cash https://cashtokens.org/docs/spec/chip/

[7] Bitcoin Cash Node - Functional Tests https://docs.bitcoincashnode.org/doc/functional-tests/

[8] Moria Functional Tests https://gitlab.com/riftenlabs/moria/moria-contract/-/tree/master/contract-tests

[9] Bitcoin Cash Research - Stable asset token - incentivized floating peg https://bitcoincashresearch.org/t/stable-asset-token-incentivized-floating-peg/1206

[10] CashScript - Smart contracts for Bitcoin Cash https://cashscript.org/

[11] Liquity https://www.liquity.org/

[12] Transaction 0b2c47c2aac582e537dbf2c50089ccacf1b34e8a610e810085a9dce86609b212 https://bch.loping.net/tx/0b2c47c2aac582e537dbf2c50089ccacf1b34e8a610e810085a9dce86609b21

[13] Rizun, P. R. (2016). "Subchains: A Technique to Scale Bitcoin and Improve the User Experience." *Ledger*, 1, 38–52. Available at: https://doi.org/10.5195/ledger.2016.40.

9 Appendix

The appendix contains the source code of the three contracts that make up the Moria protocol.

9.1 Appendix A: The Moria Contract

```
pragma cashscript ^0.10.1;

// Moria -- borrow/repay token with BCH as collateral
// Copyright 2024 Riften Labs AS

// NFT commitment in this contract:
//
// oracle sequence
contract MoriaMinter(
    bytes    oracleToken,
    bytes    loanLockingScript,
) {

    /// Trigger oracle sequence update to avoid letting borrowers
    ↪  spawn coins using old price data.
    ///
    /// This path is also taken as part of repayLoan in the loan
    ↪  contract.
    ///
    /// Design decision:
    /// Ideally; we should have two update methods; one for
    ↪  updating oracle and a second one for
    /// adding tokens; however, we hit the opcode limit so these
    ↪  path need to be merged.
    ///
    // Inputs: 00-moria, 01-oracle
    // Outputs: 00-moria, 01-oracle, 02-change (optional)
    function update() {

        // Oracle must be in this offset + 1 (both in and out)
        // (Check oracle contract itself for other output
        ↪  constraints)
        int constant index = this.activeInputIndex;
        require(index == 0);
        require(tx.inputs[index + 1].tokenCategory ==
        ↪  oracleToken);
```

```
    // Verify our output & update it with latest sequence
    bytes constant oracleMessage = tx.inputs[index +
    ↪  1].nftCommitment;
    int constant oracleSeq =
    ↪  int(oracleMessage.split(28)[1].split(4)[0]);
    int constant contractSeq =
    ↪  int(tx.inputs[0].nftCommitment);
    require(oracleSeq >= int(contractSeq));

    bytes newCommitment = bytes(oracleSeq);
    require(tx.outputs[index].lockingBytecode ==
    ↪  tx.inputs[index].lockingBytecode);
    require(tx.outputs[index].tokenCategory ==
    ↪  tx.inputs[index].tokenCategory);
    require(tx.outputs[index].nftCommitment ==
    ↪  newCommitment);
    require(tx.outputs[index].tokenAmount >=
    ↪  tx.inputs[index].tokenAmount);
    require(tx.outputs[index].value == 1000);

    // DoS prevention:
    // To avoid this utxo being flooded with dummy
    ↪  updateSequence request;
    // ensure it's not a noop.
    // Flooding the utxo can be an attack vector to remove
    ↪  the possibility of repaying a loan;
    // additionally user can mitigate getting liquididated by
    ↪  adding more collateral.
    // TODO: Set a minimum loan; this need to be set both
    ↪  here and when creating a loan.
    require((oracleSeq > int(contractSeq)) ||
    ↪  (tx.outputs[index].tokenAmount >=
    ↪  (tx.inputs[index].tokenAmount + 100 /* 1 USD */)));


    // Allow a pure BCH change output to the user.
    if (tx.outputs.length > 2) {
        require(tx.outputs[index + 2].tokenCategory == 0x);
    }

    // Because we have minting capability, we need to make
    ↪  sure there are no
    // other outputs with our token.
    require(tx.outputs.length <= 3 /* should be index + 2 + 1
    ↪  */);
}
```

```
// Borrow token with BCH as collateral
// Inputs: 00-moria, 01-oracle, +++
// Outputs: 00-moria, 01-oracle, 02-loan, 03-borrowed-tokens,
↪  04-change
function borrow(bytes borrowerPKH) {
    // Oracle must be in this offset + 1 (both in and out)
    // (Check oracle contract itself for other output
    ↪  constraints)
    int constant index = this.activeInputIndex;
    require(index == 0);
    require(tx.inputs[index + 1].tokenCategory ==
    ↪  oracleToken);

    bytes oracleMessage = tx.inputs[index + 1].nftCommitment;
    int constant oracleSeq =
    ↪  int(oracleMessage.split(28)[1].split(4)[0]);

    // Verify its not an old message.
    int constant contractSeq =
    ↪  int(tx.inputs[index].nftCommitment);
    require(oracleSeq >= contractSeq);

    int constant oraclePrice =
    ↪  int(oracleMessage.split(32)[1]);
    require(oraclePrice > 0);

    /// Calculate borrow amount
    //  We can borrow value equal to second input, minus
    ↪  collateral and fees (including required dust).
    int constant collateral = tx.outputs[2].value;

    // To ensure others have enough funds to liquididate the
    ↪  loan; limit the borrow size
    require(collateral <= 100 bitcoin);

    // collateral is 10% greater than maxBorrowBase
    int constant maxBorrowBase = ((collateral * 10) / 11);

    int constant maxBorrow = (maxBorrowBase * oraclePrice) /
    ↪  100000000;
    int constant actualBorrow = tx.outputs[index +
    ↪  3].tokenAmount;
    require(actualBorrow <= maxBorrow);

    /// Ensure contract stays alive.
```

```
/// Verify that oracle sequence is updated and borrowed tokens are subtracted.
require(tx.outputs[index].lockingBytecode == tx.inputs[index].lockingBytecode);
require(tx.outputs[index].tokenCategory == tx.inputs[index].tokenCategory);
require(tx.outputs[index].nftCommitment == bytes(oracleSeq));

require(tx.outputs[index].tokenAmount == tx.inputs[index].tokenAmount - actualBorrow);
require(tx.outputs[index].value == 1000);


// Second output is the oracle (enforced by the oracle contract)

// (no checks for oracle output (index+1))

// Third output is the collateral + an NFT representing the borrowed token amount.
// Encode this as p2sh
// This is the locking script of 'loan.cash' contract

bytes constant moriaToken = tx.inputs[0].tokenCategory.split(32)[0];
require(tx.outputs[index + 2].tokenCategory == moriaToken);
require(borrowerPKH.length == 20);
require(tx.outputs[index + 2].nftCommitment == bytes(borrowerPKH) + bytes(actualBorrow));
require(tx.outputs[index + 2].tokenAmount == 0);
require(tx.outputs[index + 2].lockingBytecode == loanLockingScript);

// Third output are the borrowed tokens
require(tx.outputs[index + 3].value == 1000);
require(tx.outputs[index + 3].tokenCategory == moriaToken);
require(tx.outputs[index + 3].nftCommitment == 0x);
require(tx.outputs[index + 3].tokenAmount == actualBorrow);

// Fourth output is pure BCH change.
require(tx.outputs[index + 4].tokenCategory == 0x);
```

```
        // We don't allow more outputs because we have minting
        ↪  capability
        // and don't want extra NFT mints.
        require(tx.outputs.length == 5);
    }

}
```

9.2 Appendix B: The Loan Contract

```
pragma cashscript ^0.10.1;

// This is a utxo holding a moria loan.
// Copyright 2024 Riften Labs AS
//
// This contract assumes and requires that its input has a NFT
↪  commitment in the
// following form:
// [borrowerPKH, borrowedTokenAmount]
//
//

contract TokenLoan() {

    // Allow adding more collateral.
    function addCollateral(pubkey borrowerPubKey, sig signature)
    ↪  {
        int constant i = this.activeInputIndex;

        // DoS prevention:
        // Add a minimum requirement to make sure the borrower
        ↪  cannot stop themselves
        // from getting liquidated by dusting the utxo.
        int constant minAdded = 100000 sats;

        // This signature check can be removed if we need to
        ↪  optimize the contract size,
        // making the contract "anyone-can-add-collateral".
        bytes borrowerPKH =
        ↪  tx.inputs[i].nftCommitment.split(20)[0];
        require(hash160(borrowerPubKey) == borrowerPKH);
        require(checkSig(signature, borrowerPubKey));
```

```
    require(tx.outputs[i].value >= tx.inputs[i].value +
    ↪  minAdded);
    require(tx.outputs[i].lockingBytecode ==
    ↪  tx.inputs[i].lockingBytecode);
    require(tx.outputs[i].tokenCategory ==
    ↪  tx.inputs[i].tokenCategory);
    require(tx.outputs[i].nftCommitment ==
    ↪  tx.inputs[i].nftCommitment);
    require(tx.outputs[i].tokenAmount ==
    ↪  tx.inputs[i].tokenAmount);
}

// Repay the loan to the Moria contract -- freeing the BCH.
//
// Design decision:
// This should ideally be split into two methods (Repay loan
↪  for owner; and repay loan for others);
// but we hit the BCH opcode limit if we create a method for
↪  it in moria.cash that doesn't require oracle input.
//
// There are two paths to repay the loan.
// - If collateral is high enough, only the loan taker can
↪  repay it.
// - If collateral is less than 10%; anyone can claim it.
//
// Inputs: 00-moria, 01-oracle, 02-loan (+ more extra for
↪  fee)
// Outputs: 00-moria, 01-oracle, 02-bchoutput
function repayLoan(
    pubkey borrowerPubkey,
    sig ourSigOrDummySig,      // User signature, or dummy
    ↪  data if collateral is too low.
) {
    int constant loanIndex = this.activeInputIndex;
    require(loanIndex == 2);
    int constant moriaIndex = loanIndex - 2;
    int constant oracleIndex = loanIndex - 1;

    // Verify that the original contract is in input 0
    require(
           tx.inputs[moriaIndex].tokenCategory
        == tx.inputs[loanIndex].tokenCategory + 0x02 /* other
        ↪  contract has minting capability */);
```

```
        bytes borrowerPKH, bytes borrowedAmountBin =
        ↪  tx.inputs[loanIndex].nftCommitment.split(20);
        int constant borrowedTokens = int(borrowedAmountBin);

        // The moria (at moriaIndex) contract verifies that the
        ↪  oracle is correct.
        bytes constant oracleMessage =
        ↪  tx.inputs[oracleIndex].nftCommitment;

        // Verify its not an old message.
        // Sequence must be greater/equal to sequence encoded in
        ↪  the contract.
        int constant oracleSeq =
        ↪  int(oracleMessage.split(28)[1].split(4)[0]);
        int constant contractSeq =
        ↪  int(tx.inputs[0].nftCommitment);
        require(oracleSeq >= int(contractSeq));

        // Check if the borrowed amount has enought collateral
        int constant oraclePrice =
        ↪  int(oracleMessage.split(32)[1]);
        require(oraclePrice > 0);
        int constant collateral = tx.inputs[loanIndex].value;
        // collateral is 10% greater than maxBorrowBase (must be
        ↪  same as in moria.cash!)
        int constant maxBorrowBase = ((collateral * 10) / 11);
        int constant maxBorrow = (maxBorrowBase * oraclePrice) /
        ↪  100000000;

        if (borrowedTokens <= maxBorrow) {
            // If the loan has enought collateral, only its user
            // can reclaim it.
            //
            // Otherwise, anyone who pays back the borrowed
            ↪  amount
            // can claim it, including collateral.
            require(hash160(borrowerPubkey) == borrowerPKH);
            require(checkSig(ourSigOrDummySig, borrowerPubkey));
        }

        // Moria contract must get their borrowed tokens back.
        // Other constrains are enforced by the moria contract.
        require(tx.outputs[moriaIndex].tokenAmount ==
        ↪  tx.inputs[moriaIndex].tokenAmount + borrowedTokens);
    }
}
```