Lava Loans Protocol

Lava Global Inc

May 12, 2023

Abstract

Serving a role as a digital store of value, bitcoin is a strong candidate for loan collateral. However, there exist no good options for users that wish to use native-bitcoin as collateral for loans without trusting a custodian. We present a solution that adapts methods from discreet log contracts to enable people to borrow assets against native-bitcoin as collateral. One of the most important use-cases of the Lava Loans Protocol is that it enables people to borrow stablecoins against native-bitcoin in the most trustless and transparent way currently possible.

1 Background

People want to borrow against their bitcoin for many reasons: to maximize exposure to bitcoin, to minimize tax liability, and to arbitrage the difference between the borrow rate and bitcoin appreciation rate. Since borrowing is almost always over-secured, people have to give up custody of their collateral to centralized parties when borrowing, and thus borrowers end up having no guarantees on how their collateral is stored or if it is even being stored. As a result, we see billions of dollars in user funds have been lost by custodians who took excess risk. It has been shown that custodians can and will most likely lose user funds. People need more transparent and secure ways to interact with financial primitives for bitcoin and crypto.

Whereas people still want to borrow, they now want solutions that are more trustless and transparent. Specifically, borrowers have the following objectives:

1. **Trust-minimized, atomic initiation of the loan.** There should be no point where the lender holds the collateral and can choose not to give the user the loan amount.

2. **Trust-minimized loan repayment.** Borrowers should be confident that if they repay their loan, their loan repayment will be accepted and that they will receive their collateral back.

3. **Trust-minimized loan liquidation.** All over-secured loans are callable, meaning the collateral can be liquidated and sent to the lender. Borrowers want confidence that they will not be liquidated unfairly, meaning that they will only be liquidated if the collateral value actually goes below a certain loan-to-value threshold determined at loan initiation.

4. Under the case of loan expiration, where the loan term closes, borrowers want confidence that they will receive their collateral back minus what is owed to the lender.

5. Borrowers desire low interest rates and fast, easy user experiences.

Lenders are also a key part of any borrowing system, and until now, they have not been able to lend against native-bitcoin collateral without either taking custody of user funds or lending against wrapped bitcoin. Lenders have the following objectives:

1. Lenders want to make sure in an over-secured loan system that the collateral is instantly callable. This is to ensure that the lender is taking the minimal risk, essentially only underwriting technology risk and liquidity risk.

2. Lenders want to ensure they can earn a high enough yield for their risk assessment, and they also want access to liquidity to remove funds if necessary.

Out of any borrowing solution that currently exists, the Lava Loans Protocol best meets the objectives of the borrowers and lenders while creating a radically faster, more trustless, more transparent, and more secure user experience.

2 Introduction

Let's start by defining some terms:

1. "Loan capital asset" is synonymous with a borrowed asset.
2. r is the compounded-daily interest rate at which the loan is issued.
3. T is the number of days after loan initialization that the loan will expire.
4. t is the number of days that have transpired since loan initialization.
5. V is the amount of bitcoin collateral.
6. L is the borrowed amount denominated in the loan capital asset.

Let $X(t)$ be the exchange rate between bitcoin and the loan capital asset at time t–this value is provided by a set of oracles. The loan-to-value ratio is defined as the following:

$$\mathrm{LTV}_{V,X,L}(t) = \frac{V \cdot X(t)}{L \cdot \left(\frac{r}{365}\right)^{365}}$$

Recognize that this accrues interest with respect to the loan capital asset. The "staked amount" is computed as the following:

$$S = L \cdot \left(1 + \frac{r}{365}\right)^T - L$$

This is the total amount of interest the lender could earn over the course of the loan. The purpose of this staked amount will be clearer in section 3.3.3.

3 Model

1. Borrower and lender are matched.
2. Loan is initialized with bitcoin collateral locked on layer 1 and borrowed asset redeemed by the borrower.
3. Contract Execution:
   (a) If the loan-to-value ratio rises too high (i.e. $\mathrm{LTV}(t) > 1-\epsilon$ for small ϵ), then the collateral is immediately redeemable by the lender. This requires the oracles to attest to a liquidation-inducing price of bitcoin.
   (b) User pays the borrowed asset back, including interest before the loan period ends to a smart contract, revealing a secret that will enable the unlocking of collateral.
   (c) If the user fails to pay back the loan before the specified end date, the loaned amount with accrued interest will be redeemable by the lender and the excess collateral will be redeemable by the borrower. This requires the oracles to attest to a price of bitcoin that determine how much is owed to both parties.
   (d) If all else fails (oracles do not attest to the price of bitcoin).

3.1 Matching the borrower and lender

Refer to section 7 for details about matching borrowers and lenders.

3.2 Initializing the contract

The process of contract initialization is extremely akin to that of atomic swaps. The borrower and lender come with bitcoin collateral and loan capital, respectively. Whereas the custody of the assets is indeed swapped with atomic swaps, in the context of lending, the collateral must instead move to a specified multi-signature address between the lender and the borrower.

To initiate this "swap", the borrower sends the collateral loan amount V to a P2WSH address with the following redeemscript where `pubkey_b` and `pubkey_l` are the borrower's and lender's respective public keys and `H(X)` is a SHA256 hashvalue whose preimage is known by the borrower.

```
opcodes                  | stack after execution
                         |
                         | 0 <sig_1> <sig_2> <preimage>
2                        | 0 <sig_1> <sig_2> <preimage> 2
OP_SWAP                  | 0 <sig_1> <sig_2> 2 <preimage>
OP_SIZE                  | 0 <sig_1> <sig_2> 2 <preimage> <size>
OP_SWAP                  | 0 <sig_1> <sig_2> 2 <size> <preimage>
OP_SHA256                | 0 <sig_1> <sig_2> 2 <size> <hash>
H(X)                     | 0 <sig_1> <sig_2> 2 <size> <hash> H(X)
OP_EQUAL                 | 0 <sig_1> <sig_2> 2 <size> 1|0
OP_IF                    |
    pubkey_b             | 0 <sig_1> <sig_2> 2 <size> <pub_1>
    pubkey_l             | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2>
    64                   | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 64
    1                    | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 64 1
OP_ELSE                  |
    pubkey_b             | 0 <sig_1> <sig_2> 2 <size> <pub_1>
    pubkey_b             | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2>
    0                    | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 0
    locktime             | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 0 <locktime>
OP_ENDIF                 |
OP_CHECKSEQUENCEVERIFY   | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 64|0 1|<locktime>
OP_DROP                  | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 64|0
3                        | 0 <sig_1> <sig_2> 2 <size> <pub_1> <pub_2> 64|0 3
OP_ROLL                  | 0 <sig_1> <sig_2> 2 <pub_1> <pub_2> 64|0 <size>
OP_EQUALVERIFY           | 0 <sig_1> <sig_2> 2 <pub_1> <pub_2>
2                        | 0 <sig_1> <sig_2> 2 <pub_1> <pub_2> 2
OP_CHECKMULTISIG         | true|false
```

At a high level, this redeemscript defines two valid execution path cases:

1. **Cooperative case:** the borrower and the lender provide signatures and a valid 64-byte preimage to the hashvalue, `H(X)`.
2. **Timeout case:** the borrower provides his signature twice with an empty preimage after a specified locktime.

The borrower signs a transaction that spends from the UTXO locked by the above redeemscript to a location where the collateral will live for the duration of the loan. This new location is a standard 2-of-2 multisig between borrower and lender from which contract execution transactions (CETs) regarding liquidation, repayment and expiration can be constructed. He sends this signature to the lender as his commitment to the lender's ability to move the bitcoin from escrow to permanent collateral location if the lender obtains knowledge of preimage `X`.

After the lender receives the aforementioned signature from the borrower, the lender will place the $L + S$ (loan + staked) amount in a smart contract on the alt-chain (e.g. Solana) that will enable the borrower to redeem L amount upon revealing the preimage `X`. The staked amount S will remain in the contract. Once the borrower redeems the loan amount and reveals the preimage on the alt-chain, the lender will be able to use the preimage to execute the cooperative path and move the bitcoin collateral to the multisig. Because the lender only has the borrower's signature to move to the multisig, the lender can only move the bitcoin to said specified location.

If the borrower does not "accept" the loan by revealing the preimage on the altchain contract, the lender will be able to recoup his $L + S$ amount after a specified timeout. Likewise, if the lender never moves the escrowed bitcoin via the cooperative path, the borrower will able to reclaim his bitcoin via the timeout path after a specified locktime.

At this point, loan initialization is complete: the collateral has ended up in a 2-of-2 multisig and the loan capital is fully owned by the borrower. The lender's stake mentioned in several paragraphs also exists in the alt-chain contract.

3.3 Contract Execution

Contract execution transactions (CETs) spend from this multisig and fall into one of four cases: liquidation, repayment, expiration and refund. These CETs are constructed in the same way as in discreet log contracts. To learn more about discreet log contracts, check out the specification here. Out of the specification, it is important to understand ECDSA-Adaptor, NumericOutcomeCompression, MultiOracle and Transactions. The rest of the paper assumes knowledge of these.

3.3.1 Liquidation

Fix a time t and reasonably small ϵ. If liquidation occurs when

$$\begin{aligned} LR(t) &< 1 + \epsilon \\ \frac{C \cdot X(t)}{L \cdot \left(1 + \frac{r}{365}\right)^t} &< 1 + \epsilon \\ \implies X(t) &< \frac{(1+\epsilon) \cdot L \cdot \left(1 + \frac{r}{365}\right)^t}{C} \end{aligned}$$

implies that $X'(t) = \frac{(1+\epsilon) \cdot L \cdot \left(1 + \frac{r}{365}\right)^t}{C}$ is the maximum asset-denominated exchange rate at which liquidation occurs on the day t.

For each day, there is a liquidation CET that yields all of the collateral to the lender. The borrower's signature is encrypted behind an adaptor signature. If the set of oracles attests to a price of bitcoin below $X'(t)$, then the lender will be able to decrypt the adaptor signature into the borrower's signature and broadcast the liquidation CET.

The lender now needs to reclaim his staked amount S (the purpose of which will be explained in the next section). The lender takes the signature for the liquidation, `XOR`s it with a specified "perturbed preimage" for the particular day and uses that value as the valid preimage for unlocking his stake. The process of verifying the ability to do this is done in section 3.2.

3.3.2 Loan Repayment

To initiate repayment, the borrower sends the principal plus interest accrued to the alt-chain smart contract. This alt-chain smart contract virtual machine must be expressive enough to assert that the correct amount has been repaid, given the time elapsed since the start of the loan.

Let's assume the lender cooperates. He "accepts" repayment of the loan by revealing a preimage to a SHA256 hash, which the borrower can use as the lender's signature for the DLC multisig to reclaim his collateral on bitcoin.

It is important that the lender cooperates when the borrower attempts to repay the loan—if he chooses not to, he could allow the loan to expire and accrue extra interest that the borrower does not need to pay. Thus, the lender must put up some stake to compel himself to work with the borrower. This stake should be equal to the amount of interest the borrower would pay back if he held the loan out to expiry, that is S as defined in the introduction. When the lender reveals the preimage, he will receive both the staked amount S and the principal + interest that the lender paid back.

3.3.3 Loan expiration and termination

At loan expiration, because no loan repayment has occurred, bitcoin is due to both the borrower and lender. Specifically $Q := L \cdot \left(1 + \frac{r}{365}T\right)/X(T)$ and $V - Q$ is due to the lender and borrower respectively.

This is paid out to the lender and borrower exactly as a traditional DLC would. An important parameter we make native to the loans protocol is the "attested price bucket size". This is the size of buckets of bitcoin price values that share a single CET. If this value is 5000, then a single CET would be constructed for the attested price of bitcoin in the range 0-5000, 5001-10000, 10001-15000, etc. Typically this is set to a much smaller value. By increasing this value, we decrease the granularity of the contract execution, implying that fewer adaptor signatures need to be created, verified and stored. Intuitively, the time and space complexity of adaptor signature creation/verification and storage is inversely related to the "attested price bucket size". That is, if we double the value of of "attested price bucket size", we should halve our time and storage usage.

Furthermore, if the crosses a timelock threshold (i.e. expires) on the alt-chain, then the lender should be able to redeem his stake amount S on the alt-chain smart contract.

3.3.4 Refund

If all else fails and none of the three execution events occur, then the borrower is entitled to redeem all of his collateral after a locktime (e.g. 2 after normal expiration).

The lender will likewise be able to claim his staked amount S for the same reason mentioned in the previous subsection on loan expiration.

4 Optimization with Zero Knowledge Proofs

An astute reader may have noticed that the two cross-chain behaviours mentioned above are not clearly "trustless". First, why should the borrower trust the preimage that the lender reveals on the altchain during repayment acceptance is indeed the lender's signature to the repayment CET? Second, why can we XOR the bitcoin liquidation signature with an associated piece of information to get a singular preimage for stake reclamation? There is nothing that immediately suggests this–in fact, hash functions are designed not to expose any useful information about the preimage.

To address these concerns, the protocol specification employs two ZK-Starks.

4.1 Loan Repayment

In the context of loan repayment, we wish to know that the preimage to a certain hashvalue is in fact the signature of a bitcoin transaction given by a private key. In the following example, `signature` is the ECDSA secp256k1 signature of a bitcoin transaction, `msg_hash` is the message that is being signed (i.e. the bitcoin transaction), and `pubkey` is the secp256k1 public key of the lender.

```
func signature_commit(
    signature: Signature,
    msg_hash: Message,
    pubkey: PublicKey,
) {
    assert verify_ecdsa_signature(sig, msg_hash, pubkey)
    let hashed_signature = sha256(sig)
    commit([hashed_signature, msg_hash, pubkey])
}
```

The lender will run the above program and create a proof that he ran the program with the committed outputs, `hashed_signature`, `msg_hash` and `pubkey`. When the borrower verifies the proof, he can feel confident that `hashed_signature` is in fact the `sha256` hash of a signature of a transaction that will enable the borrower to reclaim his collateral.

4.2 Liquidation

Recall that for the lender to reclaim his stake on the altchain after a liquidation, he needs to take the borrower's bitcoin signature for the liquidation transaction and `XOR` it with a specified perturbed preimage.

```
func liquidation_signatures_commit(
    sigs: Vector<Signature>,
    msg_hashes: Vector<Message>, // assume same lengths
    length: int,
    pubkey: PublicKey
) {
    let preimage: [byte; 64] = [0; 64];
    for i in 0..length {
        assert verify_ecdsa_signature(sigs[i], msg_hashes[i], pubkey)
        preimage ^= sigs[i]
    }

    let hash = sha256(preimage)

    let perturbed_preimages = [[0; 64]; length];
    for i in 0..length {
        perturbed_preimages[i] = preimage ^ sigs[i]
    }

    commit([hash, msg_hash, pubkey, perturbed_preimages])
}
```

The borrower runs the above program in the zero-knowledge virtual machine and creates a proof that he ran the program with the committed outputs. When the lender verifies the proof, he will be confident that he can take the borrower's bitcoin signature for a particular liquidation transaction (after its decrypted from the adaptor signature by the oracles attestation), `XOR` it with a perturbation preimage indexed by the day of liquidation and use that resulting computation as a preimage that will unlock their stake on the alt-chain.

5 Oracle Considerations

There are several parameters regarding the oracles that must be selected ahead of time:

- **The set of oracles:** the specific oracles in charge of the attestation of the price of bitcoin.
- **Oracle threshold:** the number of oracles that must agree for a liquidation or expiration contract execution event to occur. For instance, if the set of oracles has size 5 and the oracle threshold is 3, then any three of the five oracles must agree.
- **Oracle error tolerance:** the extent that oracles must generally agree. This parameter is actually two numbers, `min_support` and `max_error` where `min_support` is the minimum difference in price attestations that must be tolerated while `max_error` is the max difference that can be tolerated. Both numbers must be powers of two and `min_support` < `max_error`. The price differences in the range of the two numbers are not guaranteed to be supported.
- **Attested price bucket size:** discussed in section 3.3.3.

6 Risks

There are three notable risks worth discussion:

1. **Oracle risk.** The set of oracles could collude with either of the parties. If the oracles collude with the lender, they could cause a liquidation event to execute even if the collateral is worth well over the loan value, causing the borrower to lose the amount: collateral value - loan value. If the oracles collude with the borrower, they could prevent liquidations from occurring, attest to a high price of bitcoin at expiration such that the majority of the collateral is unduly returned to the borrower. They could also simply not attest at all and force a refund CET.

2. **Loan capital risk.** Let us say the loaned capital is a stablecoin that we realize is not backed 1:1 with dollars, and it loses its peg. Paying back the stablecoin amount becomes much easier for the borrower, and he is very inclined to do so. This is further exacerbated by the fact that that the oracles are attesting to the price of bitcoin denominated in dollars, not stablecoin (when loan expires the value of the loan in dollars is send the the lender). Consequently, the brunt of the stablecoin risk is put on the lender.

3. **Altchain risk.** If the altchain fails, then the borrower will have no way to repay his loan. This means the loan will necessarily expire, and some of the collateral will go to the lender. If the borrower had sold his stablecoin for a different asset off the altchain (say fiat US Dollars), then he is comfortable. However, because the altchain has failed, the lender will never be able to reclaim his staked amount S, which can be assumed to be lost.

While these risks exist, there are many ways to respond to them:

1. **Oracle risk.** Oracles for the Lava Loans Protocol are not natively aware that a contract is relying on their date. They are blind to contracts. Oracles also have little incentive to misbehave. They can't receive borrower/lender funds as funds only move to borrowers or lenders. Oracles are also negotiated at loan initiation so borrowers and lenders are both aware of the oracles involved in a loan contract. We can also increase the set of oracles to make the contracts as trust-minimized as possible by adding redundancy. To make this easier, Lava has released an open-source oracle implementation called sibyls.

2. **Loan capital risk.** Lenders and borrowers can decide which stablecoins they want to interact with, so they can assess risks prior to the initiation of their contracts. If a loan interfaces with USDC, USDT, or another 1:1 backed stablecoin, funds can also be recoverable via contacting Circle, Tether, or the respective custodian.

3. **Altchain risk.** Lenders and borrowers can decide what chain/network they would like the stablecoins to live on before loan initiation. Some funds can also be recovered by contacting the fiat custodian.

7 Implementation as a Dealer's Market

The maker/taker model lends itself well to peer-to-peer contracts such as these. The current implementation has the borrower as the taker and the lender as the maker. For now, the list of lender-takers is hosted by Lava such that the borrower pulls the list, chooses a lender and constructs a contract with him individually.

Creating a loan contract with a lender is a four-step process:

1. **Fetch lender terms:** retrieve the lender's (dealer's) terms to determine if he is willing to match.

2. **Fetch lender public key (knowns):** the passes in the loan details to retrieve a public key for the lender that is unique to the contract. This public key is used to create the bitcoin escrow transaction. The borrower can then broadcast the bitcoin escrow transaction and waits for it to reach 6 confirmations.

3. **Create contract:** the borrower sends the borrower details, loan details, loan capital asset id, the confirmed escrow funding txid and initialization hashvalue. The lender stores this information, and computes the repayment hashvalue and lender details. He then returns it to the borrower. Now the borrower has all the information required to construct the contract and will then create his own set of signatures.

4. **Finalize contract:** the borrower sends the escrow funding txid as a unique id for the contract along with his signatures. The lender verifies the borrower's signatures, creates the lender signatures and returns those signatures. After this, he funds the alt-chain contract and the loan begins.

The aforementioned types look are structured as the following:

```
/// The non-negotiable terms that a dealer sets
struct DealerTerms {
    /// the inclusive minimum amount of a loan in usd
    min_amount: u64,
    /// the inclusive maximum amount of a loan in usd
    max_amount: u64,
    /// a set of brackets specifying the the minimum interest rate a dealer is
    /// willing to accept for an loan-to-value ratio. must be a disjoint set
    /// that convers (0, x] where x is the maximum loan-to-value ratio the
    /// dealer is willing to accept
    ltv_interest_rate_brackets: Vec<LtvInterestRateBracket>,
    /// the loan-to-value ratio (in basis points) at which the loan will
    /// liquidate
    liquidation_ltv_bp: u16,
    /// the inclusive minimum length of a loan
    min_expiry_days: u16,
    /// the inclusive maximum length of a loan
    max_expiry_days: u16,
    /// the set of oracles
    oracles: Vec<Oracle>,
    /// the number of oracles that must agree for an contract execution event to
    /// occur
    oracle_threshold: usize,
    /// the oracle price attestation error that the dealer can tolerate.
    /// deconstructs to ‘‘(min_support_exp, max_error_exp)‘‘ where ‘‘2 **
    /// min_support_exp‘‘ is the minimum difference in price attestations
    /// that must be tolerated while ‘‘2 ** max_support_exp‘‘ is
    /// the max difference that can be tolerated. both numbers be powers of 2
    /// and ‘‘min_support_exp < max_support_exp‘‘
    oracle_error_tolerance: (usize, usize),
    /// this denotes the size of the buckets of bitcoin price values that share
    /// a single cet for particular contract execution events
    attested_price_bucket_size: usize,
    /// details regarding fees to a middleman, say a host of a dealer registry
    protocol_fee_options: Option<ProtocolFeeOptions>,
    /// bitcoin network fee rate per vbyte in basis points
    network_fee_rate_per_vb_pp: Option<u16>,
}

struct LoanDetails {
    /// The amount of bitcoin collateral in satoshis
    collateral_amt_sat: u64,
    /// The amount of loan capital being borrowed in this contract. This value
    /// is denominated by a unit associated with each supported loan capital
    /// asset. For USD loans, this unit is commomly one dollar, implying
    /// ‘loan_capital_amt=10000‘ corresponds to a $10,000 loan
    loan_capital_amt: LoanCapitalAmount,
    /// Loan capital denominated interest rate in basis points
    interest_rate_bp: u16,
    /// The official start date of the loan
    start_date: Date,
    /// How many days until the loan expires and force closes
    loan_expiry_days: u16,
}

/// Public details about the borrower
struct BorrowerDetails {
    /// The borrower’s public key
    fund_pubkey: PublicKey,
    /// The script pubkey where the borrower will receive his change (i.e.
    /// excess inputted bitcoin than needed for collateral)
    change_script_pubkey: Script,
    /// The script pubkey where the borrower will receive his collateral when
    /// the contract closes
    payout_script_pubkey: Script,
    /// Material to construct borrower’s address in loan capital contract
    loan_capital_address_material: Vec<u8>,
}

/// Public details about the lender
struct LenderDetails {
    /// The lender’s public key
    fund_pubkey: PublicKey,
    /// The script pubkey where the lender will receive the borrower’s
    /// collateral if he is due any
    payout_script_pubkey: Script,
    /// Material to construct lender’s address in loan capital contract
    loan_capital_address_material: Vec<u8>,
}
```

The following two structures are a part of the loan details:

```
/// Details regarding fees to a middleman, say a host of a lender registry
struct ProtocolFeeOptions {
    /// The script pubkey where the origination fee will be paid to
    origination_fee_script_pubkey: Script,
    /// The script pubkey where the interest fee will be paid to
    interest_fee_script_pubkey: Script,
    /// Origination fee of the loaned amount in basis points
    origination_fee_bp: u16,
    /// Fee on interest in basis points
    interest_fee_bp: u16,
    /// Material to construct fee receiver’s address in loan capital contract
    loan_capital_address_material: Vec<u8>,
}

/// Represents the minimum interest rate a lender-dealer is willing to accept
/// for an loan-to-value ratio in a range
struct LtvInterestRateBracket {
    /// the inclusive loan-to-value ratio range for which this interest rate is
    /// valid
    ltv_ratio_range_bp: (u16, u16),
    /// the minimum interest rate in basis points valid for this given
    /// loan-to-value range
    interest_rate_bp: u16,
}
```

8 Conclusion

We present this protocol to bring more trustless and transparent functionality to bitcoin. The technology we use to build the protocol can be adapted to build other functionality that uses native-bitcoin. Bitcoin collateral can also live on lightning, not just layer 1. Lava believes the future of sovereign finance will involve sovereign functionality being built for sovereign money.