Notebook:
Accountable Identity

Nathaniel Masfen-Yan, Solal Afota, Dhruv Mangtani,
Sacha Arroues-Paykin

July 2022

1 Abstract

Accountable identity is something we encounter everyday. If we commit a crime, the police know what we look like, where we live, and therefore can arrest us. When interacting with financial services in Web2, we must comply with KYC regulations to be held accountable in case we commit fraud. These laws keep us safe, but for 99% of people who act according to the law, they come at the cost of privacy. Web3 has created a world where users can remain anonymous, our information isn't sold to the highest bidder, and governments don't have supreme control. However, this has come at the cost of accountability and trust. DeFi has become a hub for financial crime, Opensea estimates 80% of NFT projects are fraudulent, and many ICOs have enriched few at the cost of many. This lack of trust in Web3 has been felt strongly in the past couple of months. This paper introduces Notebook, a novel protocol with Sybil-resistant log-in, credential aggregation, and Self-Sovereign identity that allows for accountability and trust while preserving anonymity and privacy. As a team, we believe in decentralization and privacy. Therefore, we have made it our mission to make Web3 safe so that it can become universal.

2 Introduction

Notebook provides users with a set of fragmented identities that have the following properties:

1. Users can prove their humanity
2. It is impossible to link these identities together (unless they have the user's secret key)
3. It is impossible to link these to the user's real-world identity
4. Credentials can be aggregated across identities

5. Each human only receives a single set of fragmented identities

To our knowledge, no other identity solution offers these properties. Furthermore, this set of properties has many applications such as trustful governance, safe NFT marketplaces, and fully anonymous credit scoring. This paper also details an implementation of Notebook using the Circom and Solidity languages on the Ethereum Virtual Machine.

On blockchains where users identify themselves through wallet addresses, identities are disposable and replaceable, leading to a critical breakdown in trust. Protocols have no protection against users committing fraud and re-identifying themselves with a new address; therefore, they resort to other forms of identification when they require trust. These forms of identification, such as KYC, often breach users' anonymity by forcing them reveal information about themselves.

The public nature of blockchains means that users often have to resort to using multiple addresses to protect their anonymity. Furthermore, users will use different wallets when interacting with various chains and ecosystels. Since their activity is split between addresses, users have to reveal them to aggregate their credentials (i.e. prove that they have contributed to 10 DAOs), breaching their anonymity.

Applications such as under-collateralized loans and credit scoring require these properties. Users must aggregate their credit score across all their wallets, and must not create a new set of identities if they default on a loan (Sybil-resistance). This application is explored below in more detail.

3 The Notebook Protocol

3.1 Overview

Notebook begins by getting users to upload identifying information to an audited third party to verify their humanity and that they have not previously created a Notebook. If a user is validated, a hash of their secret key with a random bit string is signed. This signature attests to their humanity and the uniqueness of their account. To provide Sybil-resistant log-in, a user will use this signature to add $H(sk||r)$ to a Merkle tree. They can later prove, in Zero-Knowledge, an opening to the tree. The user will create four fragmented identities by hashing their secret key with a counter. Credentials (i.e. credit scores) for each fragmented identity are stored in Merkle trees. Zero-Knowledge proofs are used to aggregate a user's credentials across their fragmented identities. Users can also interact with authenticated parties to log data to a specific fragmented identity, impacting their aggregate credentials. Users can also add attested credentials to a Merkle tree allowing them to have a provable Self-Sovereign identity.

3.2 Variable Definitions

We define by $H()$ the Hash function MiMC-7 that is assumed to be collision resistant and pre-image secure. We chose MimC-7 for its computational efficiency and compatibility with SNARKs. We define sk as a 256-bit secret used to demonstrate ownership over an identity and its credentials. We define sk_W as a 256-bit secret which acts as their wallet password. We define by r a 256-bit random seed generated by the user. Each user has 4 sub-identities they will use to interact with protocols. The audited third-party that verifies a user's humanity has an ECDSA key pair (sk_N, pk_N). We define by $\mathcal{C}$ a contract that can verify ZK-SNARKs and is responsible for keeping an up-to-date set of Merkle roots (discussed in the implementation section). We define the Sybil tree as a Merkle tree of depth 32, with leaves of the form $L = H(sk||r)$ where leaves are allocated to users on a first-come-first-serve basis as users onboard. For each score that a user has (i.e. credit score, reputation score), there is a Scoring tree of depth 256. The value at index $H(sk||i)$ will store the value of the user's i-th fragmented identity score. Furthermore, we want to prevent a malicious actor from adding their identity multiple times to the *Sybil* tree as this could lead to an attack in the case where they invalidated their Notebook (discussed below). Therefore we store a $Created$ map on $\mathcal{C}$.

3.3 Creating a Notebook

Notebook has an initial onboarding phase which acts as a proof of humanity check. The users firsts generate a secret sk and a random nonce r. A user then submits an ID and a short video of their face, which is used to create a face mask. A hash of the face mask is stored to ensure that if the same user attempts to create a second Notebook, a collision would accur the user would be denied. The user uploads a leaf, L, to be signed and provides a ZK-Proof to the signing party that they know sk, r such that $L = H(sk||r)$. This check ensures that the user is acting according to the protocol definition. If the check passes, $H(sk||r)$ is signed by the audited third party using the ECDSA key sk_N. We denote this signature $Signature$. Next, the user must add this identity to the *Sybil* Merkle Tree stored on the smart contract $\mathcal{C}$. First, we must check that the user hasn't already added this identity to the *Sybil* Merkle Tree. Then we verify that $Signature$ is a valid ECDSA signature of $H(sk||r)$ with public key pk_N. So, we first check that L is not in the created map. Then, the user will provide an Update Merkle proof to the next available leaf of the *Sybil* tree using the following SNARK.

$$SNARK(H(sk||r), \pi, MerkleRoot, j)$$

The proof will check that π is a valid update proof for $H(sk||r)$ being added to index j. If the proof is valid, the smart contract updates the stored $MerkleRoot$ and increments j.

3.4 Fragmented identity

Once the onboarding process is complete, the user is completely anonymous: there is no way to link their Notebook back to their real-world identity. However, attacks exist in which attackers map a user's Web3 activity by tracking their wallets allowing them to create profiles. These can sometimes be used to link wallets to users' real-world identities. Since privacy and anonymity are core values of Notebook, we strongly encourage our users to use multiple wallets to prevent these attacks. Therefore each user creates 4 sub-identities, each linked to a different wallet address. These identities are

$$H(sk||1), H(sk||2), H(sk||3), H(sk||4)$$

It is up to the user to decide how they want to segment their identities.

3.5 Sybil Resistant Login

In many cases, a protocol may want a user to prove ownership over a Notebook. For example, they may want to check that the user is not a bot, or that they have not already registered an account.

The user will present a proof that they know sk, r such that $H(sk||r)$ is a leaf of the *Sybil* Merkle tree. To do this, they will use an Opening proof. The user also needs to prove that they haven't previously logged in through another sub-identity. To do this, they will provide $Null = H(sk||ProtocolAddress)$. The protocol will store the value in a map such that if the user created another account, there would be a collision.

$$ZK - SNARK(sk, r, L = H(sk||r), \pi, Null, ProtocolAddress)$$

Here we have that sk, r, L, π are private inputs to the proof. The proof works in the following way:

1. Check that $L = H(sk||r)$
2. Check that π is a valid opening for $H(sk||r)$ in the Sybil tree.
3. Check that $Null = H(sk||ProtocolAddress)$

The protocol then checks that $Null$ hasn't already been used, and if it hasn't, it stores $Null$ and lets the user log in.

3.6 Key Recovery and Identity Theft

We entrust our users to store their secret keys as they would store the private keys of their wallets. If the user loses their private key, our protocol supports BIP-39 allowing the user to enter their seed phrase and recover their key. If the user were to lose both their secret key and their seed phrase, we have support for

the social recovery of their key. If the user's secret key is stolen, they will quickly notice due to the transparency of the protocol. Furthermore, they will want to invalidate their current Notebook. Due to the inherent nature of accountable identity, we have to defend against malicious actors claiming to have a stolen identity due to a possible tainted Notebook. Therefore there is a 6 month delay where the user will not have access to a Notebook. To get a new Notebook, the user will undergo the following process:

1. The user will submit a proof of ownership of the Notebook and signal to the smart contract $\mathcal{C}$ that they are invalidating their Notebook. We must first verify the Update proof of the *Sybil* Merkle Tree used to change the value of the leaf where $H(sk||r)$ is stored to 0. If the proof is valid, the smart contract will update the Merkle root stored, and an event will be emitted on-chain signalling that the identity has been anulled.

2. Next, the user will take a face mask and prove ownership of the Notebook to the onboarding servers. The servers will first check that there is a collision, checking that the user did have a Notebook. Then the server will check that the ownership proof is valid. Then the server will listen for the event emitted by the smart contract $\mathcal{C}$ to verify that the user has indeed invalidated their Notebook.

3. After a 6-month wait period - enforced by the onboarding servers - the user can create a new notebook by going through the onboarding process again.

3.7 Accountable Identities

With Notebook, a user has 4 different identities, each associated with four credentials:

- Fraud flag - this boolean value signifies whether the identity was reported for being part of some fraud (like operating a pump-and-dump scheme).
- DAO score - this represents the activity and contributions of a user in decentralized governance.
- Social Reputation - this is a value which represents a user's engagement in communities and forums. A high score would signify meaningful participation, whereas a lower score may demonstrate that a user harmed the communities they were a member of.
- Credit Score - this reflects a user's engagement in DeFi.

Individual protocols and communities can also petition for other credentials to fit their needs. The smart contract $\mathcal{C}$ stores the $MerkleRoot$ of a tree for each piece of information. Each tree is 256 layers deep. A user's identity $H(sk||i)$ will be the index of the leaf where their information is stored. When a user

interacts with a protocol through one of their identities, they may give the protocol permission to write to one of these trees. For example, if a user joins a new community, it may request to write to their Social Reputation if the user breaks the community rules. However, since the user has 4 different identities, we need a way to aggregate their credentials. Otherwise, the user could be malicious on one identity and change identities. Furthermore, users' good actions can be aggregated: if the user uses multiple identities for their DeFi / GameFi activity, they can aggregate their credit score to reflect the full extent of their activity. This is what we mean by *Accountable* identity. A user's reputation should persist between different addresses. However, the user's identities should not be linked when aggregating their credentials. A user can prove their aggregated credentials using the following proof

$$ZK - SNARK(sk, r, L = H(sk||r), \pi_{Sybil}, O_1, \pi_1, O_2, \pi_2, O_3, \pi_3, O_4, \pi_4)$$

Here (O_i, π_i) is an opening and its proof for a leaf of the Merkle Tree. We must do an opening for each identity. Here we need the following inputs $(sk, L, \pi_{Sybil}, O_i, \pi_i)$ to be secret. Otherwise, it would be possible for a malicious actor monitoring the execution of the contract to link the identities together. The proof has the following steps:

1. Check that π_{Sybil} is a valid opening proof for L in the $Sybil$ tree.
2. Check that $L = H(sk||r)$
3. For each opening O_i, We check that its index is of the form $H(sk||i)$.
4. For each identity, verify that the proof of the opening O_i is correct.
5. Aggregate and output the scores

3.8 Permissioning

When interacting with a service, they may request to write to a user's identity. To enable this, we store the $MerkleRoot$ of the $Permission$ Merkle Tree on our smart contract $\mathcal{C}$. To give the user complete control over their data, they can allow a smart contract with address $\mathcal{A}$ to write to a specific credential $info$. To do this, the user adds a leaf with value $H(\mathcal{A}||info||H(sk||i))$ to the $Permission$ Merkle tree (of depth 32), allowing the protocol to write to $info$ of the identity $H(sk||i)$. The mechanism by which a user's actions on a protocol are reflected in their score, works similar to a liquidation: if certain conditions are met, a third party can call a 'log score' function, paying the gas to write a log and receive some financial compensation. This third party would provide an opening to the $Permission$ tree to show that the protocol is authorized to write to a user's data.

3.9 Self-Sovereign Identity

Self-Sovereign identity works through a system of verified credentials. Users will get a piece of information I verified and signed by a third-party organisation using an EdDSA key pair s_T, p_T. To do this, the user will provide $Cred = H(sk||I)$ to be signed along with a Zero-Knowledge Proof that $Cred = H(sk||I)$ (where sk is a hidden input). We denote $Cred_S$ as the signature of $Cred$. First, we must verify that $Cred_S$ is a valid EdDSA signature of $Cred$. Then, a user can add this piece of information to the first available leaf of the SSI Merkle Tree, which is 32 layers deep, using the following proof:

$$ZK - SNARK(sk, r, L, \pi_{Sybil}, Cred, \pi_{SSI})$$

Where sk, L, π_{Sybil} are private inputs. The proof has the following steps

1. Check that $L = H(sk||r)$
2. Check that π_{Sybil} is a valid proof for the opening L in the Sybil tree
3. Check that π_{SSI} is a valid update proof to add $Cred$ to the SSI tree
4. Update the Merkle Root of SSI

If a user wants to prove piece of information I about themselves, they can use the following Opening Proof for the SSI tree

$$ZK - SNARK(sk, I, Cred, \pi)$$

Where sk is a secret input. The proof has the following steps:

1. Check that $Cred = H(sk||I)$
2. Check that π is a valid opening for $Cred$ in the SSI tree.

At any time, a user can nullify a credential by providing a valid opening for the SSI tree of the credential they wish to annul, followed by an Update Proof to change the value of the leaf to 0.

4 Security Analysis

Any polynomial-time adversary who doesn't know sk or sk_W should have a negligible probability of linking a user's fragmented identities. For the security of our protocol, we rely on the following assumptions.

1. MIMC-7 is Collision Resistant
2. MIMC-7 is Preimage Secure
3. ECDSA & EdDSA Signatures are unforgeable
4. The KYC party is curious but follows the protocol
5. Completeness, Soundness, and Zero-Knowledge of ZK-SNARKS

5 Implementation

5.1 Notebook as a Wallet

We have developed Notebook as a wallet to allow users to seamlessly link their addresses to their sub-identities. A user will have a separate secret key, sk_W, which will serve as the secret key to their wallet. Furthermore, addresses will be paired, by the user, to their fragmented identities. To preserve user anonymity, transactions between these four addresses will be routed through Tornado cash. Furthermore, before making a transaction, web-scraping tools will be used to warn the user about the risks a given transaction poses to their anonymity.

5.2 Notebook as an Optimistic Rollup

The presence of Merkle Tree proofs with multiple hidden inputs induces high gas fees for multiple functionalities described in the protocol. Therefore, we have built Notebook as an Optimistic Rollup with non-interactive proofs to lower fees by $\approx$ 20x. On a high level, this means that a proof can be verified off-chain, and then published on chain, allowing independent validators to check them.

Suppose a user wishes to interact with Notebook. The user will create the relevant proof (for a tree opening, update, etc.) and has two options. Either the user can submit the proof to $\mathcal{C}$, which will verify it on-chain and act accordingly, or the user can send the proof to a Notebook server. The former case will incur higher gas fees but must be possible as a security precaution. In the latter case, the server will verify the proof off-chain, if valid, there are two possibilities:

1. The proof is an opening proof or an aggregation proof that doesn't change the state of any Merkle trees. In this case, the server stores the proof in the $Proof$ queue and then calls $\mathcal{C}$ with the address of the user (A_U) and any relevant details as arguments. $\mathcal{C}$ will store A_U in the $Address$ queue on-chain. The Notebook server has a whitelisted address A_{NB}, which means $\mathcal{C}$ does not need to verify the proof and will act as if it is valid.

2. The proof is an Update proof that changes the Merkle trees. In this case, the server also stores the proof in the $Proof$ queue and calls $\mathcal{C}$ with the arguments $Proof$, A_U, and the new state (ie. updated Merkel roots). Firstly, $\mathcal{C}$ will store A_U in the $Address$ queue. It will then check that the $Proof$ and $Address$ queues have the same length, as otherwise the server cheated. If this check passes, it will service the user's Update proof. Next, it will calculate a Merkel tree root where each leaf is $H(Proof||State||A_U)$ in the case of an opening proof and $H(Proof||State||NewState||A_U)$ for the Update proof. Here we use the Keccak-256 hash function due to its low gas cost and the fact we don't need SNARK compatibility. Furthermore, if the Notebook server followed the protocol, $\mathcal{C}$ can easily pair $Proof$ with A_U for each leaf as they will be in the same index position in the $Proof$

and $Address$ queues. $\mathcal{C}$ will store the Merkle tree root in the $StateHistory$ list, set the $Address$ queue to empty, and store the new state.

If a user submits a valid Update proof directly on-chain, $\mathcal{C}$ will update the state after a time delay. This delay allows the Notebook server to submit all the opening proofs it may have already processed. The user's Update proof and address will be added to the $Proof$ and $Address$ queues, and $\mathcal{C}$ will follow the same procedure as in bullet point two above.

Suppose $\mathcal{C}$ is a smart contract on an ecosystem with token $TOKEN$. The Notebook server will stake some $TOKEN$ on an Escrow smart contract. Since all calls to $\mathcal{C}$ are public, independent validators have access to all the proofs, and their corresponding addresses, submitted by the Notebook server. Furthermore, validators have access to the history of Merkle roots. Therefore, validators can verify every proof off-chain. Since each proof contains a reference to the wallet address submitting it, the Notebook server cannot take a valid proof and submit it under a different A_U to $\mathcal{C}$. If a validator finds a faulty proof (tuples $(Proof, State, A_U)$ or $(Poof, State, New_State, A_U)$), then they can submit a fraud-proof to $\mathcal{C}$. For this, they first show that $H(Proof||State||A_U)$ or $H(Proof||State||New_State||A_U)$ is a leaf of a Merkle tree whose root is in the $StateHistory$ list. Next, $\mathcal{C}$ will verify the disputed proof. If the proof is false, half of the staked $TOKEN$s are burnt, and the rest is sent to the validator.

5.3 Onboarding

An AWS EC2 instance is authenticated to grab the signing key from an HSM cluster via KMS. The instance runs an Apache web server hosting a Flask app with an endpoint for signing $H(sk||r)$ during onboarding. Signing takes place after a trusted third party does a proof of humanity. The unhashed user's identity is never forwarded to the EC2 instance or made visible to Notebook services; it is captured and signed solely through the third party. The identity hash is forwarded from the third party to the EC2 instance and stored in an S3 bucket for collision checking.

5.4 Merkle Tree Updates

When a Merkle Tree is updated, $\mathcal{C}$ emits an event onto the Ethereum Mainnet. Each Merkle tree is represented by a subgraph on The Graph Network: a decentralized protocol that continually indexes Ethereum data into subgraphs per their manifests. While the entire set of leaves of the $Permission$ tree is stored as a subgraph, it is infeasible to do the same for the scoring tree of depth 256. Thus, only a sparse representation containing the nonzero leaves is stored. The hashes of null trees of depths 1-256 are precomputed and stored directly in the subgraph query function for fast, optimized root computation.

6 Application: Credit Score Aggregation

In the traditional economy, credit score is a metric based on multiple parameters such as age, revenue, payment history, etc that evaluates a borrower's trustworthiness. In the context of DeFi, most of these metrics are unavailable, but a credit score metric is still relevant. This paper will focus on implementing a credit score compatible with credit score aggregation.

6.1 Why Aggregate Credit Scores?

For Alice to take a loan from one of her wallets, she'll need to prove that her aggregate credit score is in a given range (this range will determine how advantageous the loan's conditions are for her). This way, honest borrowers can enjoy the practicality of using multiple wallets without compromising their credit score while defaulting borrowers are punished across the line and not just through a single wallet's credit score.

The advantage of ZK-proving that Alice's credit score is in a range, and not directly providing said credit score, is for privacy:

Given $S(w_1), ..., S(w_nk)$ the scores of the wallets of n users each having k wallets, and Alice's exact credit score, not many $k-$tuples of wallets could correspond to Alice's: a malicious third-party could accumulate more data on Alice than initially possible. Only providing a range containing Alice's credit score gives out far less information (since many more $k-$tuples could correspond to her wallets), and is therefore much more secure.

6.2 Objectives

We'll abusively use the term "take a loan" meaning "taking a loan and successfully, or not, repaying it". We want to implement two things:

1. To each wallet w assign a credit score s, updated each time said wallet takes a loan. We'll denote S the function that fetches a wallet's credit score, and given a wallet w and a loan l, and denote Δ the function such that $\Delta(w, l)$ is w's credit score variation due to the loan l

2. To each person, say Alice, owning multiple wallets $w_1, ..w_n$, we want to associate an aggregate credit score easily obtainable from the credit scores $S(w_1), ..., S(w_n)$. In this paper we will choose the aggregate to be $\sum_{i=1}^{n} S(w_i))$

Furthermore, when Alice takes a loan, we don't want the global credit score variation to depend on the wallet the loan is taken from : if $w_1, ..., w_n$ are Alice's wallets, and l is a loan, we want

$$\forall i, j \in \{1, .., n\}^2, \Delta(w_i, l) = \Delta(w_j, l)$$

This eliminates any arbitrage ability for the borrower. Aggregate credit score only depends on the loans taken.

6.3 Price Oracles

When updating a wallet's credit score after a loan l, we inevitably have to access the amount borrowed, fees, interest paid, etc. Since loans are taken in various currencies, we need a price oracle to convert the borrowed amount into a unique stablecoin (we chose USDC), used for credit score calculation. However, price oracles are subject to manipulations, which could severely affect credit score calculation. Hence, we decided to opt for Uniswap v3's TWAP price oracle: by providing the average exchange rate over a given period, it highly increases the cost of oracle manipulations, making them, in our case, completely impractical. Furthermore, the main disadvantage of such oracles is reduced accuracy during volatility. However, in practice, accurate values of one's credit score are not used, as protocols rely on the proof that one's credit score is in a given range.

6.4 Platform Verification

Updating the credit score of a wallet w given a loan l requires some knowledge of the platform the loan was taken on. Otherwise, a malicious borrower could collude with an unscrupulous lender for an arbitrarily large credit score boost. We mitigate the problem by manually verifying some of the main platforms: good behavior is rewarded only on verified platforms, whereas bad behavior is punished similarly everywhere.

6.5 The Δ function

We are now faced with a rather simple problem to solve: assuming one of Alice's wallets, w, took a loan l, how should w's credit score evolve? Let's name a few things :

- $V \in \{0; 1\}$, is 1 if the lending platform is verified, 0 otherwise
- b, the borrowed amount
- c, the collateral amount
- i, interest meant to be paid.
- f, amount of fees paid.
- $p \in \{True, False\}$, whether the contract was respected or not.

The following function respects the wanted conditions :

- If p, ie the borrower was honest, then

$$\Delta(w, l) = V(f + i)$$

- If $\neg p$, ie the borrower defaulted, then

$$\Delta(w, l) = f - i + min(c - b, 0)$$

This way, the credit score stores the amount a user has paid, minus the amount defaulted. w's exact credit score $S(w)$ is known, and is updated :

$$S(w) \leftarrow S(w) + \Delta(w, l)$$

6.6 How can an AMM use the Aggregate Credit Score?

We will suggest slight improvements to AAVE protocol which manage risk by calculating a loan's LTV (Loan to value, ie $\frac{Value(Borrowed)}{Value(Collateral)}$) and LT (liquidation threshold). On a given platform, Alice can, without providing any information, request to borrow a set of assets of value at most equal to the value of $\sum_i (Collateral_i \cdot LTV_i)$ when depositing collateral of $\sum_i Collateral_i$. The LT of a given collateral wallet is the weighed average of the LT's of the individual assets : $LT = \frac{\sum_i Value(Collateral_i) \cdot LT_i}{\sum_i Value(Collateral_i)}$. When the value borrowed raises above the collateral value times the LT, the loan is liquidated.

6.6.1 Potential Improvement

A credit score reflects how much a borrower has spent on loans and fees across the DeFi ecosystem. A platform can therefore afford to offer riskier contracts to high credit score borrowers. Similarly, borrowers with a low (negative) credit score are bigger risk factors: LT and LTV can be adjusted to further reduce risk compared to the initial parameters.

Let's assume a platform is willing to risk 1% of what an individual has spent in fees plus interest, ie $\frac{c}{100}$ USDC for an individual with a credit score c. Initial parameters give an LTV of LTV_0 and an LT of LT_0. If c is the value of the collateral then $b = c \cdot LTV$ is the maximum value of the borrowed assets. In order to increase the borrowed amount for the same collateral, and keep liquidation conditions the same, we multiply both LT_0 and LTV_0 by $1+\epsilon$ (ie, the borrowed amount is $b(1+\epsilon)$ with same collateral). This means the amount left to reimburse after liquidation goes from $b' - c' \cdot \alpha$ to $b' \cdot (1+\epsilon) - c' \cdot \alpha$, with b' the borrowed asset's value at liquidation, c' the collateral's value at liquidation and α the discount factor during liquidation. By setting ϵ such that $\epsilon \cdot b < Value(\frac{c}{q100} USDC)$ with $q > 1$, and liquidating the loan whenever $b' > q \cdot b$, the platform can never lose more than $\frac{c}{100}$USDC while providing the borrower with less collateralised loans. Similar principles apply to negative credit scores: multiplying LT_0 and LTV_0 by $1 - \epsilon$ for $\epsilon > 0$, a platform can reduce the maximum default amount and therefore reduce risks for untrustworthy borrowers.

References

[1] AAVE Whitepaper V2, 2020. https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf.

[2] R. S. Alexey Pertsev, Roman Semenov. Tornado Cash Privacy Solution, 2019. https://tornado.cash/Tornado.cash_whitepaper_v1.4.pdf.

[3] M. S. R. K. D. R. Hayden Adams, Noah Zinsmeister. Uniswap v3 Core, 2021. https://uniswap.org/whitepaper-v3.pdf.

[4] G. Konstantopoulos. How does Optimism's Rollup really work?, 2021. https://research.paradigm.xyz/optimism.

[5] A. V. S. B. Marek Palatinus, Pavol Rusnak. Mnemonic code for generating deterministic keys, 2013. https://en.bitcoin.it/wiki/BIP_0039.

[6] C. R. A. R. T. T. Martin Albrecht, Lorenzo Grassi. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity, 2016. https://eprint.iacr.org/2016/492.pdf.

[7] J. P. Yaniv Tai, Brandom Ramirez. The Graph: A Decentralized Query Protocol for Blockchains, 2018. https://github.com/graphprotocol/research/blob/master/papers/whitepaper/the-graph-whitepaper.pdf.