Notebook: A Zero-Knowledge Identity Infrastructure Layer

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

Abstract

This technical whitepaper presents Notebook, a novel protocol for Sybil-resistant log-in and credential aggregation that preserves user anonymity. Notebook provides users with a set of fragmented identities with the following properties: with each identity, users can prove they are a human; the identities are detatched from one another; the identities are detatched from the user's real-world identity; credentials can be aggregated across identities and each human only receives a single set of fragmented identities. This set of properties has many applications in fully anonymous credit scoring and governance. This paper also details an implementation of Notebook using the Circom and Solidity languages on the Ethereum Virtual Machine.

1 Problem Statement

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 the majority 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

In order to protect their anonymity on public blockchains, it is already commonplace for users to 'fragment' their identity and use different wallet addresses for different activities. Notebook allows users to continue to protect their anonymity by fragmenting their identity whilst ensuring the following properties:

1. Users can prove their humanity with any fragmented identity
2. It is impossible to link these identities together (unless the user's secret key is leaked)
3. It is impossible for any third party or adversary to link a fragmented identity to the user's real-world identity

4. Credentials can be aggregated across identities
5. Each human receives a single set of fragmented identities

This set of properties has many applications in trustful governance, safe NFT marketplaces, and fully anonymous credit scoring. This paper details an implementation of Notebook using the Circom and Solidity languages on the Ethereum Virtual Machine.

3 Overview

Notebook lets users prove both on and off-chain credentials and aggregate reputation scores across their fragmented identities. There is a single SSI tree of depth 36 for all users. Each reputation score is a tree of depth 256, where users are allocated leaves through a hash function.

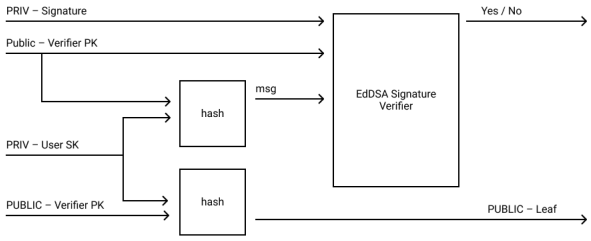

Figure 1: Architecture for Adding a Private Credential

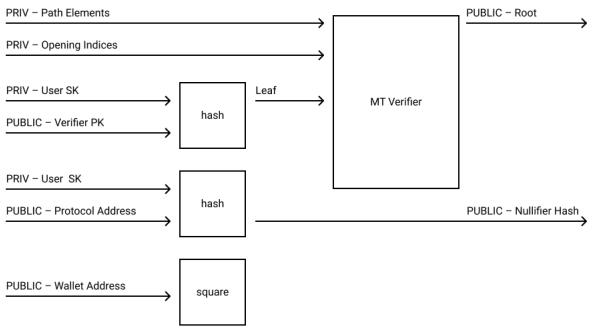

Figure 2: Architecture for Proving a Credential

4 Adding a Credential to Notebook

Self-Sovereign identity works through a system of verified credentials. Users will get a credential verified and signed by a third-party organisation using an EdDSA key pair s_T, p_T. The schema of this is outlined in figure 1. The user provides $Cred = H(sk||p_T)$ to be signed along with a

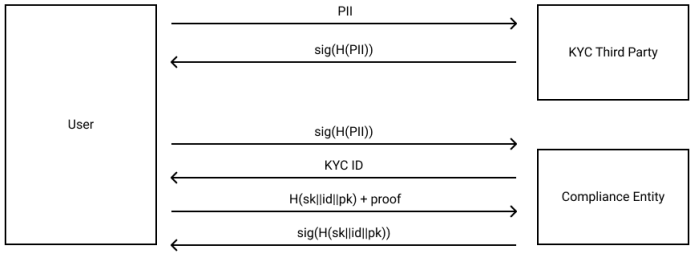

Figure 3: Architecture for Getting a KYC Credential Signed

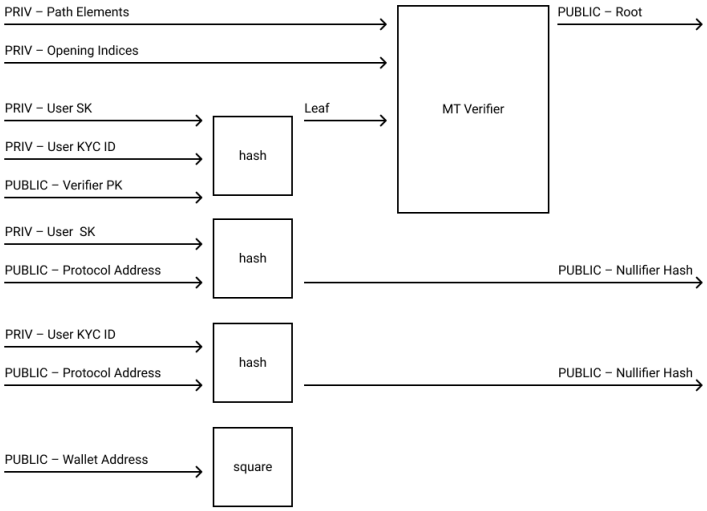

Figure 4: Architecture for Proving a KYC Credential

Zero-Knowledge Proof that $Cred = H(sk||p_T)$ (where sk is a hidden input). Then in order to add their credential on-chain, a verifier checks in zero-knowledge that the signature is correct, and adds $H(p_T||sk)$ to the SSI tree of depth 36. The leaf is added to the tree in Solidity (not in a Zero-Knowledge proof), and is a different bit string than the message that was originally signed, ensuring that the credential verifier is unable to breach the user's anonymity. An event is then emitted on-chain with the leaf's value and index.

Because the pre-image of the leaf contains both the signer's public key and the user's Notebook key, membership of the leaf in the tree proves that a given user owns a certain credential.

5 Proving a Credential

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 architecture for this proof is shown in figure 2. A user proves that they own a leaf in the SSI merkle tree of the required credential. The opening proof is given as a secret input, preventing the

protocol or any other party from learning when else that credential was used or by what other wallet it was used by. The proof may also output a nullifier hash if the protocol requires that an owner of a given Notebook can only create one account. The user's wallet address is also inputted into the proof to prevent malicious frontrunning.

A user can prove ownership over a credential from multiple wallets without linking them, given that this proof never exposes any information about the user's Notebook, besides that it owns the stated credential.

6 KYC

In order to fulfill KYC regulations, if a user commits fraud, there must be a way to link their PII (private personal information) to their wallet address. In the model where a user submits PII to a KYC verifier, receives a signature, adds a credential to a tree with a wallet and then accesses the credential with different wallets, there must be a way for trusted third parties to collude to link these pieces of information.

In order to comply with these restrictions, we introduce another party, called the compliance entity, which has the power to link the wallets a user uses a given KYC credential with. This entity can't link a user's PII to their KYC credential and needs to collaborate with the the KYC verifier to establish this link. The role of this entity is to decide when to reveal the link between a user's wallet address and their PII based upon the needs of regulators. Initially this party will be the Notebook Labs team, but the hope is to decentralize control over this process.

The overall schema for getting a KYC credential signed is outlined in figure 3. Both the KYC verifier and the compliance entity have ECDSA keys. The KYC third party knows the link between a user's PII and the hash of their PII. The compliance entity knows the link between the hashed PII and the wallet addresses which use the KYC credential. It does this by giving the user a unique 'KYC ID' when they create their KYC credential.

Each time a user uses their KYC credential, they publish a hash of their KYC ID and the protocol's address. Whilst onlookers are unable to learn anything from this hash, the compliance entity can find in $O(n)$ time, where n is the number of KYC IDs, the ID of the user which published the hash. The architecture for proving a KYC credential is outlined in figure 4.

7 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 four 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.

7.1 Accountable Identities

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

- 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

7.2 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.

8 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. Poseidon is Collision Resistant
2. Poseidon is Preimage Secure
3. ECDSA & EdDSA Signatures are unforgeable
4. The Credential Verifier is curious but follows the protocol
5. Completeness, Soundness, and Zero-Knowledge of ZK-SNARKs

8.1 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.

9 Implementation

9.1 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 Merkle 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 Merkle 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 smart contract determines the proof is false, it burns half of the staked $TOKEN$s, and sends the rest to the validator.

10 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.

10.1 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.

10.2 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.

10.3 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.

10.4 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.

10.5 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.

we construct credit score to be a 128−bit number, storing both the number of defaults by a wallet (on the 64 first bits) and the total amount paid by the wallet in interest and fees, minus the amount defaulted (on the remaining 64 bits) 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

$$\Delta(w, l) = 2^{64} + f - i + min(c - b, 0)$$

w's exact credit score $S(w)$ is known, and is updated :

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

we set the base credit score of a wallet to be 2^{63}: this way, the amount paid in interest and fees minus the amount defaulted by a wallet can range between 2^{63} and -2^{63} USDC.

How can an AMM make use of the provided credit score?

To demonstrate, we will suggest slight improvements to LTV (Loan to value, ie $\frac{Value(Borrowed)}{Value(Collateral)}$) and LT (liquidation threshold) calculations, based on already existing risk parameters (AAVE).

10.6 Initial protocol

Overview of the initial parameters: 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 a 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 can be liquidated.

10.7 Possible modifications

Let's assume Alice has an aggregate credit score of CR.

$cr = Cr \mod 2^{64}$ reflects how much Alice has spent in interest plus fees on various platforms.

Let's assume a platform is willing to risk 5% of what Alice has spent in fees plus interest, ie $\frac{cr-2^{63}}{20}$ USDC, as long as $cr - 2^{63} > 0$ and Alice has never defaulted on one of her loans.

Initial parameters give an LTV of LTV_0 and an LT of LT_0. If c is the value of Alice's collateral, then $b = c \cdot LTV$ is the maximum value of the assets Alice can borrow.

When multiplying LT_0 and LTV_0 by $1 + \epsilon$ (ie now, Alice can borrow $b(1 + \epsilon)$ with the same collateral), liquidation happens rigorously at the same moment as with the initial parameters. The situation is analog to one with initial parameters, where Alice provides collateral of value $c(1 + \epsilon)$, except Alice is spared from providing $c \cdot \epsilon$ additional collateral.

By setting $\epsilon = \frac{cr-2^{63}}{c \cdot 20}$, in the case of stablecoin collateral (otherwise, the minimum amount lost compared to the base parameters isn't bounded), we have:

- If Alice respects the lending contract, the platform earns $(1+\epsilon)$ times more interest, and Alice can still benefit from less collateralized loans.
- In the case of a default, the platform loses at most $c \cdot \epsilon = \frac{cr-2^{63}}{20}$ compared to the initial parameters, but Alice loses all benefits on said platform.

The same can be done for negative credit scores: multiplying LT_0 and LTV_0 by $1 - \epsilon$ for $\epsilon > 0$, platforms reduce risk by raising collateral percentage, for all currencies.

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] OpenSea. NFT Fraud, 2022. https://twitter.com/opensea/status/1486843204062236676.

[8] 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.