zkFi: Privacy-Preserving and Regulation Compliant Transactions using Zero Knowledge Proofs

June 2023

Amit Chaudhary
amit@zkfi.tech
amit.chaudhary.3@warwick.ac.uk

Abstract

We propose a middleware solution designed to facilitate seamless integration of privacy using zero-knowledge proofs within various multi-chain protocols, encompassing domains such as DeFi, gaming, social networks, DAOs, e-commerce, and the metaverse. Our design achieves two divergent goals. zkFi aims to preserve consumer privacy while achieving regulation compliance through zero-knowledge proofs. These ends are simultaneously achievable. zkFi protocol is designed to function as a plug-and-play solution, offering developers the flexibility to handle transactional assets while abstracting away the complexities associated with zero-knowledge proofs. Notably, specific expertise in zero-knowledge proofs (ZKP) is optional, attributed to zkFi's modular approach and software development kit (SDK) availability.

1 Introduction

Achieving privacy in blockchain applications presents unique challenges - often requiring trade-offs between user experience and privacy. The transparent nature of conventional blockchains reveals all of the transaction data, including addresses, assets involved, amount, smart-contract data, and timestamps, out to the public. It is analogous to using a regular bank account and revealing all private financial information, deterring the mass adoption of blockchain and digital asset technology.

As this space continues to evolve and more institutional and individual users engage in activities on these applications, privacy will become a paramount concern creating the biggest hurdle for achieving mainstream adoption. Individuals contemplating the adoption of blockchain-based payment systems may exhibit considerable hesitance if their salaries or other confidential financial details, such as payments for medical services and their online purchases, are accessible to the public. This demand for privacy will also be from social networking platforms, decentralized lending protocols, philanthropic platforms, e-commerce, gaming, and other protocols where users want to prioritize safeguarding the privacy of their information.

While there is a clear need for privacy solutions, regulatory scrutiny of privacy protocols necessitates action to develop practical and fair measures that deter bad actors from engaging in on-chain illicit activity. **Selective De-anonymization**, as mentioned in [1], lays out a method for allowing traceability. Particularly, an instantiation of **involuntary de-anonymization** as practically studied in [2] can prove to be a flagship regulation-compliant technique that can be used when a malicious actor refuses to comply with the law.

In this paper, we propose a privacy-preserving solution with solid regulatory compliance using zero-knowledge proofs and threshold cryptography having the following features:

- A general purpose **multi-chain** privacy solution spanning across multiple EVM chains.
- Available with **simple, composable and flexible plug-and-play middleware** solution via an SDK.
- Secure with **built-in compliance** solution with concrete AML practices.

- Providing a seamless **user-experience** using account abstraction and wallet integrations [4, 5].

2 Limitations in current architecture

At present most widely used programmable blockchains (e.g. EVM based chains such as Ethereum, Polygon, Optimism, Arbitrum) offer benefits such as permissionless nature, decentralization, and security, but these blockchains do not offer privacy. Alternative blockchain networks have been aiming to create solutions from scratch, to eliminate the pitfalls but fail to near the activity and value of mentioned public chains. This necessitates a solution to multiple problems on the public chain itself.

Lack of Privacy. A regular on-chain transaction exposes private data and transactions to the public. The data such as sender/receiver address, asset type, and quantity, smart-contract data, timestamps, etc. are conveniently available in an organized manner to the general public through block explorers such as Etherscan [3]. This information can be used to track funds for targeted attacks, identify users, and extract sensitive information and patterns about their activity. These pitfalls prevent the adoption of revolutionary blockchain applications by several serious users, especially users like institutional investors and businesses.

Weak Compliance. Privacy problem implicitly poses another severe issue of compliance. How to have robust compliance in place and prevent bad actors while maintaining user privacy? Enabling privacy on decentralized platforms has been very well known to attract malicious actors abusing the platform. These include using it for illicit activities like laundering stolen funds from hacks or preparing for one. Most of the time, these actors have succeeded because of the lack of firm AML (Anti-Money Laundering) practices to deter bad actors. Weak compliance deters institutional investors or businesses from entering the blockchain space for legitimate usage.

Lack of Infrastructure. Building private applications on the blockchain has been made possible by the advancements in Zero Knowledge (ZK). However, implementing ZK technology is complicated. Developers need specialized knowledge and resource investment in ZK development. This creates overhead and distraction from the divergence of resources from developing a core of their applications.

Poor User-Experience. These distractions and overheads due to the lack of middleware privacy solutions lead to a subpar developer and user experience. Even if the application develops its privacy layer, the UI/UX faces challenges for users. While UX is still an area to be improved in web3 generally - it is even worse in the ecosystem of privacy-preserving applications.

2.1 Previous Solutions

ZCash blockchain was among the first to tackle privacy by facilitating anonymous transactions. While innovations there have been impressive, they could not reach the intended adoption nor offered any programmability - restricting only to peer-to-peer transactions. Hence, missing out on the prominent application level use-case for, eg. DeFi. **Monero**, another private chain, more or less shares the same context.

Tornado protocol on Ethereum amassed a significant number of usage despite not-so-good UX. But it overlooked compliance and became go-to-place for money laundering [7]. A portion of its volume has ties to large-scale hacks [8], attracting serious implications from regulators. It, too, only offered peer-to-peer private transactions lacking any interoperability beyond that. Aztec came up with a novel solution with their L2 roll-up approach that had DeFi compatibility to some extent. However, it required users to bridge their assets back and forth and had significant waiting times - making it not-so-practical for all kinds of DeFi interaction, e.g. in swaps because of slippage.

Others also share the same problems, especially weak compliance guarantees and friction in user experiences.

3 zkFi: A Middleware Solution

To tackle problems, as discussed above, zkFi offers a packaged solution that acts as a privacy middleware with built-in compliance. Privacy and compliance-related complexity are abstracted away by providing the developers with an SDK that facilitates a plug-and-play solution.

3.1 Privacy with Zero Knowledge Proofs (ZKPs)

zkFi uses ZKPs to facilitate privacy at its core by achieving the following goals:

- Performing private transactions concealing sender, recipient, and amount of funds being transferred.
- Ensuring transactions cannot be linked together, preventing tracking the flow of funds.
- Preventing double-spending by proving the transaction is valid without revealing information about the transaction itself.
- Selective de-anonymization by ensuring verifiable encryption of transaction data.

While there are multiple formulations of ZKP systems available and being researched, zkFi specifically utilizes a groth16 [18] zkSNARK system which is currently most suitable for on-chain privacy applications.

3.2 Stronger Compliance Guarantees

Compliance has been the conundrum for privacy protocols so far. zkFi aims to have an industry-standard compliance framework such as:

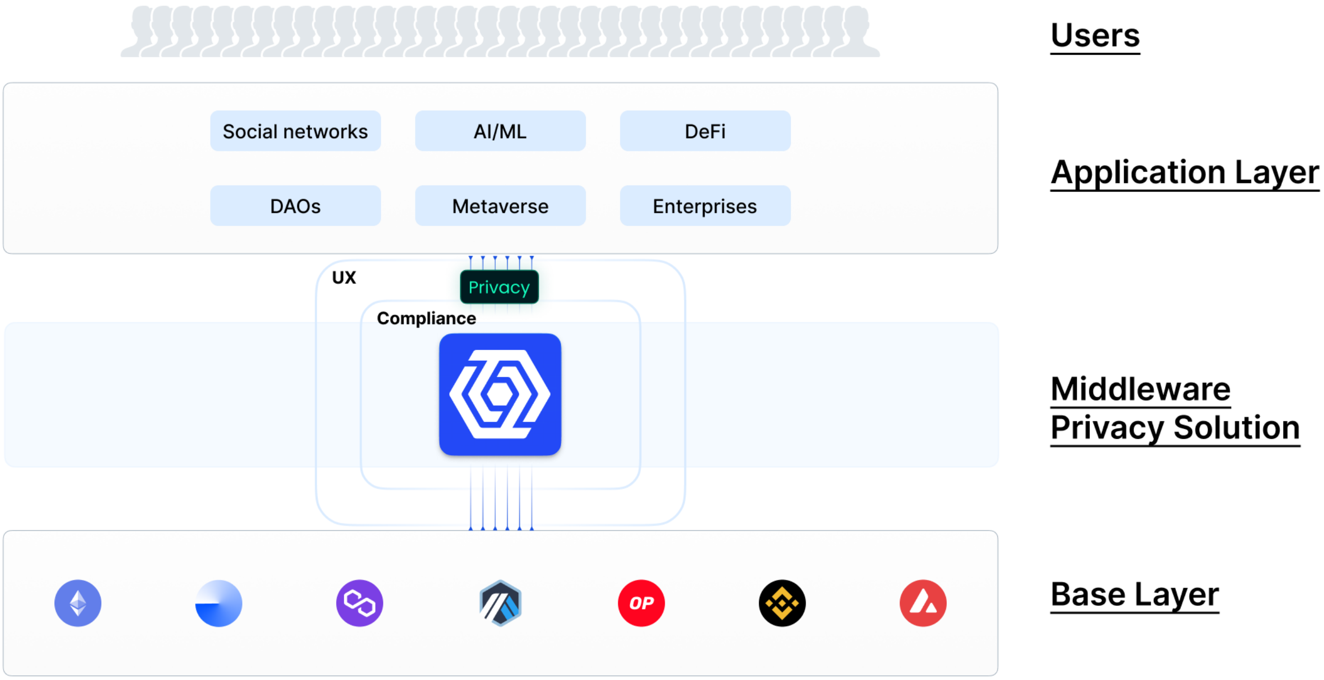

Figure 1. Middleware Solution

- **Selective De-Anonymization:** For an industry-standard AML practice, a process for the de-anonymization of user-specific private transaction data needs to be followed. It could be a **voluntary de-anonymization** where the entity in question can share general or per-transaction viewing key to a regulatory authority. In other cases (for malicious actors), **involuntary de-anonymization** may be enforced in response to a regulatory emergency or court order. The latter is an accountable multi-party process to prevent abuse of power. It is thoroughly studied in [2] as SeDe (short for Selective De-anonymization) framework for compliance and is a flagship among all compliance tools available.
- **Deposit Limits:** We put a fiat limit on the asset being transacted and/or the volume flowing through the protocol in a time period. There can be a provision to relax this limit for specific entities like a compliant businesses.
- **Risk Management and Screening** The protocol sets up compliance and risk management integrations to identify and prevent any kind of illegal financial activity. This could be achieved by services like TRM Labs[1] products and Chainalysis[2] oracles perform screening and identifying inflow of illegal funds into the system.

3.3 Pluggable Privacy With SDK

By offering an SDK, a full set of compliant privacy features is instantly available to protocols and their developers. SDK facilitates a **simple composable plug-and-play solution** that abstracts away every bit of ZKPs or compliance-specific complexities. This renders immense benefits to protocols such as:

- New protocols can focus on developing their core features without investing time and resources into ZK development.
- Existing protocols do not have to modify their smart contracts for compatibility. It's a simple plug-and-play through SDK.
- Compliance issues often come as part of a privacy conundrum. But with zkFi, protocols will not have the burden of juggling privacy-related compliance practices.

In addition to the advantages mentioned above, the **flexibility of interaction** with the integrating protocol remains at the hands of a developer. Given private user assets and/or

[1] https://www.trmlabs.com/

[2] https://go.chainalysis.com/chainalysis-oracle-docs.html

data as input, developers are free to write any custom logic to apply to the inputs.

See section 6.4 for implementation details.

3.4 Account Abstraction and UX

The advent of EIP-4337 [12] Account Abstraction proposal allowed for significant improvements in user experience across protocols. One such improvement or feature it brings is the ability for smart contracts to pay for gas fees of transactions. This allows for gas-sponsored transactions or payment of gas fees in ERC-20 tokens.

In privacy protocols, there exists a common problem referred to as *gas dilemma* or *fee payment dilemma*. A gas dilemma exists because if users need to pay the gas fees from their wallet to execute their transactions then this gas payment discloses the user's public profile since their address as the sender of the transaction is now visible publicly.

zkFi uses account abstraction features from EIP-4337 so that the gas fee can be paid in ERC-20 tokens making transactions on zkFi a smooth experience. The flexibility of gas payment is left to the integrating protocol - allowing it to sponsor transaction fees or charge from their users in any desired way. In the case of peer-to-peer transactions, a user simply pays gas in transacting assets. This is facilitated through a custom EIP-4337 **Paymaster** contract that pays the gas on the user's behalf in exchange for a small fee in any supported asset.

3.5 Wallet Integrations

By implementing a simple interface, provided within SDK, for the shielded account operations (e.g. signing private transactions), crypto wallet applications can support private transactions for their users. In that case, the shielded account keys share the same security space as the wallet's private keys. Developers simply send defined requests to the shielded account for, say, authorizing transactions by signing it. This allows developers to directly use the shielded account in their application UIs without concern about securely handling sensitive keys.

Some examples of such wallet integrations are:

- **MetaMask**: MetaMask is a browser extension as a crypto wallet. Through its **Snaps API** [4] it allows developers to extend its functionality by publishing custom packages of logic called "Snap". One such Snap, the "zkFi Snap" acts as an interface to the shielded accounts within MetaMask.
- **Ledger**: Ledger is a hardware wallet that securely stores private keys for multiple cryptocurrencies. It allows **Embedded Apps** [5] to be installed on the user's device. A zkFi app is one such app tailored to securely store shielded account keys and authorize private transactions from the device.

4 Use Cases of zkFi

Pluggable Privacy for DeFi protocols. As mentioned, previously in section 3.3, the design of the infrastructure allows any existing DeFi protocol for seamless integration to enable anonymous transactions.

For instance, consider Aave[3] which is a lending/borrowing liquidity protocol. Aave users normally supply assets to one of Aave markets and get interest-bearing tokens a.k.a *aToken*[4] in return. For Aave to let its users supply assets anonymously, it'd just require a simple stateless proxy contract, let's call it *AaveProxy*. The job of *AaveProxy* is just to take an asset and return the corresponding *aToken* asset by talking to APIs already written by Aave in their core contracts. The proxy is very loosely coupled, no ZK circuit programming is required at all nor any changes in Aave's existing contracts.

The same kind of seamless integration is possible with other protocols for - earning interest anonymously on Compound, doing anonymous swaps on Uniswap, staking anonymously on Lido, and many more.

Private Payments via Stealth Address. While normal peer-to-peer private transactions are supported, one may opt to receive assets in the form of payment to their stealth address - which preserves anonymity one can share a random-looking address each time.

So, for instance, **payment links** can be generated by a receiver and can be shared with a sender such that the shared link has no traceability to any previous payment. This is similar to sharing payment links in Stripe[5] but with anonymity.

Shielded Account for protocol UIs. Via its integration directly in existing crypto wallets, starting with MetaMask Snaps integration, any protocol may request spending of user's private assets instead of public assets from a normal wallet. This frees protocols to define their own custom UIs on their own domains and request interaction with Shielded Wallet however it wants.

This adds up with a better user experience thanks to built-in account abstraction features like gasless transactions.

zkFi strives to offer a general solution, so it is future compatible with other use cases that may arise as the need for privacy is realized more and more.

5 Architecture

The diagram in figure 2 shows an architectural overview of the system with involved actors and their interaction with each other:

Wallet Provider. The host for shielded account. This includes crypto wallets (e.g. MetaMask and Ledger) which

[3] https://aave.com/

[4] https://docs.aave.com/developers/tokens/atoken

[5] https://stripe.com/in/payments/payment-links

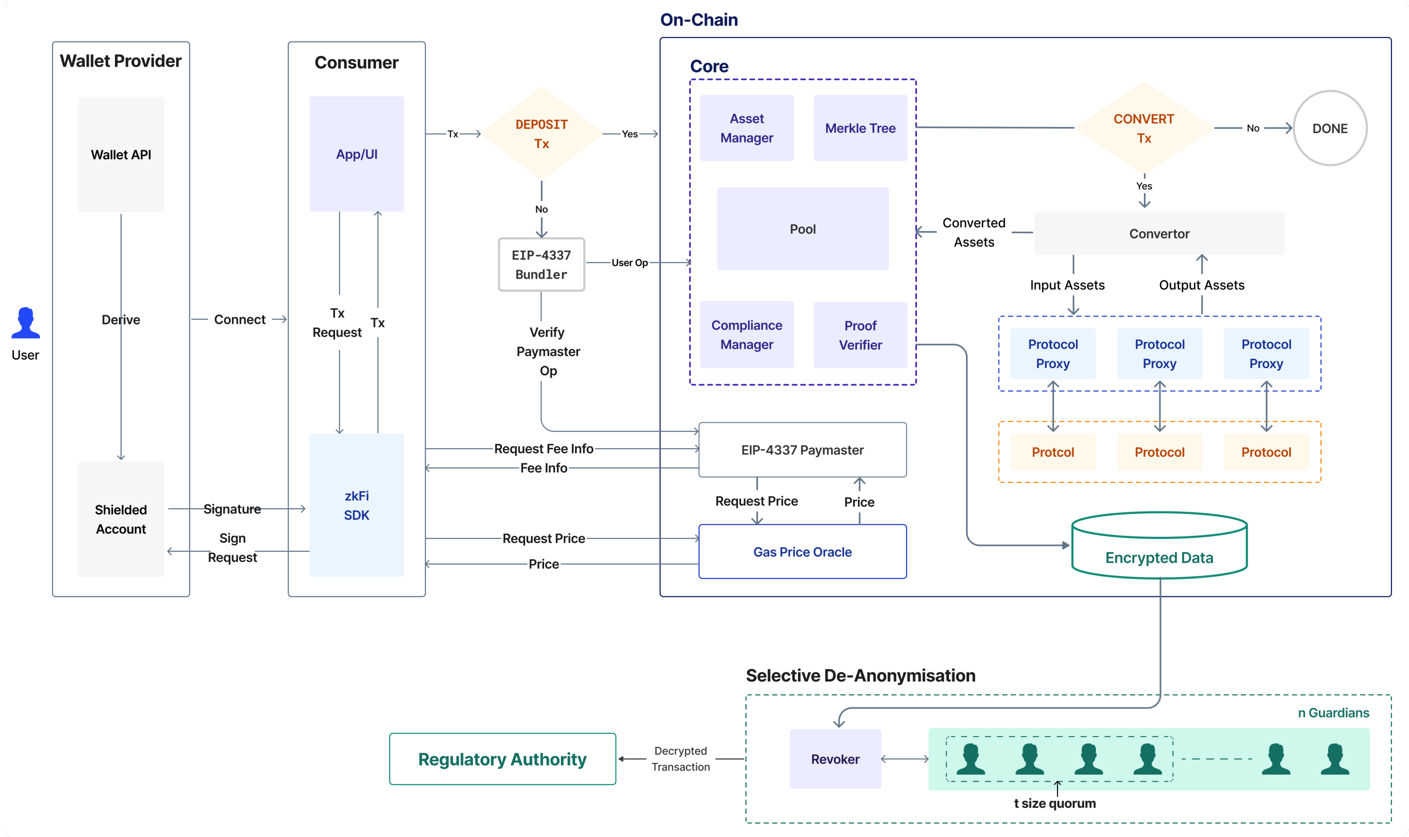

Figure 2. Architecture Diagram

provide functionalities to deterministically derive keys of shielded account from ethereum account of user.

Consumer. The consumer of the SDK that communicates with the Shielded Account via a connection to the wallet provider. It can invoke private transaction approvals by requesting transaction signature. The SDK consumers can be an application user interface or command-line applications which can construct complex transactions by passing only simple transaction requests to SDK. The SDK parses the request and handles creation, signing and (ZK) proving of the transaction. Ultimately, SDK outputs a suitable transaction payload to be sent on-chain.

EIP-4337 Bundler. A bundler sends the transactions to the network through a EIP-4337 bundler node rather than directly from a wallet. This has two main benefits - the user avoids exposing their wallet address publicly and it can pay for the gas with shielded/private assets itself.

Gas Price Oracle. A decentralized oracle to consulted by SDK and paymaster to calculate the equivalent amount of gas price in ERC-20 tokens at the time of transaction. A decentralized exchange like Uniswap [6] could be a suitable oracle.

EIP-4337 Paymaster. A custom EIP-4337 compatible paymaster that pays for the user operations in exchange for fees cut in the asset being transacted. The paymaster validates the operation after consulting Gas Price Oracle and making sure enough fee will be paid.

Core. The core of zkFi protocol encapsulates a **multi-transactional multi-asset** pool, an asset manager, a merkle tree of notes and an on-chain ZK proof verifier. See section 6.3.

Convertor. A smart contract that mediates the **convert** operation on behalf of core. It involves calling target protocol (e.g. DeFi) via its proxy for performing an operation (e.g. swap, stake, lend) and returning any resulting assets as a result of operation, back to core.

Protocol Proxy. Proxies are simple smart contracts that implement/extend a simple interface/base provided by SDK. Doing this allows it to plug nicely into the Core and be able to receive assets from it to perform any DeFi operation. See section 6.4.

Guardians. Guardians are multi-party entities that exist to perform the **involuntary selective de-anonymization**

process upon a verifiable request from a revoker entity as per the [2]. See section 6.5.

6 Building Blocks

6.1 Shielded Account

A user holding a shielded account can sign valid transactions and decrypt balances and transaction data. Unlike normal Ethereum accounts which are controlled by a single private key, a shielded account contains two types of such keys.

We assume the existence of a cryptographically secure random number generator (RNG) function used to generate private keys:

$$\mathtt{RNG}\colon \phi \longrightarrow \mathbb{B}^{256}; () \mapsto \xi \tag{1}$$

We cryptographically derive the keys of the shielded account using the poseidon hash function (from 25) and group operations on the BabyJubJub curve (see Sec. 7.2).

6.1.1 Sign Key.

The sign private key can sign valid transactions authorizing the spending of user funds in shielded accounts. A new private key is generated by first sampling an entropy element, ξ by invoking RNG and then hashing it with a fixed salt value Δ_{sign}.

$$s = \mathtt{poseidon}(\xi, \Delta_{sign}) \tag{2}$$

The corresponding public key S is simply a point on the curve as:

$$S = s \cdot G \tag{3}$$

6.1.2 View Key.

The view private key decrypts the user balances and transaction history. This private key is generated by hashing the entropy ξ with salt Δ_{view}:

$$p = \mathtt{poseidon}(\xi, \Delta_{view}) \tag{4}$$

and public key is $P = p \cdot G$.

During the construction of a transaction, p is used to symmetrically encrypt transaction data (e.g. amounts, owner) using **ChaCha20-Poly1305** encryption scheme [15]. The resulting ciphertext is decrypted and later utilized for future transactions.

Having a distinct key for read-only access renders multiple advantages including revealing p for compliance purposes without giving up spending authority and allowing protocol websites for read-only access of data to display on their custom UIs.

This also allows to have a transaction-specific view key giving the ability to reveal only selected transactions or protocol-specific viewing keys so that protocol websites only get to read data relevant to it rather than the entire transaction history.

6.1.3 Shielded Address.

The shielded address, A of a shielded account is the public key of both key pairs:

$$A \equiv (S, P) \tag{5}$$

A user may choose to post A on a public address registry for others to look it up using public wallet addresses or ENS names and conveniently send funds to address A. Although A is not directly used as is while associating funds to it in a zkFi transaction, it allows the derivation of a random-looking "stealth address", x from A. x is then used instead for the purpose.

For the sake of better user experience for crypto wallet users, the RNG at equation 1 should be facilitated by the wallet application, and the entropy ξ be seeded by the private key or seed phrase. This is possible with the wallet integrations (see Sec 3.5). In that case, the keys of the shielded account can be derived from the wallet account itself. This frees the user from handling additional keys.

6.2 Stealth Address

A stealth address is a one-time address that is randomly generated to receive funds to. This address is not linked to the receiver's permanent address. Hence making it difficult to link or track a particular user's transactions and protect their identity.

There are multiple different schemes available for deriving a stealth address from a user account. These schemes allow the receiver to detect funded stealth addresses meant for them and calculate the associated private key from auxiliary data, normally sent along with the transaction. The calculated private key can then be used to sign transactions authorizing the spending of funds at that stealth address. One example of such a scheme is defined in EIP-5564 [13].

In zkFi, we primarily use ZK proofs (but also signatures) for constructing valid transactions. This allows for a much simpler and more efficient way to generate and use stealth addresses in the protocol. A zkFi stealth address, x is derived from sign public key S as:

$$x = \mathtt{poseidon}((S, \delta)) \tag{6}$$

where δ is a random element also referred to as a blinding factor.

To relay any auxiliary data to be included in the transaction request, the sender generates an ephemeral public key, $Q = r \cdot G$ where $r \in \mathbb{B}^{248}$ is another random element.

A sender then calculates a shared key K from the view public key P of the receiver as:

$$K = r \cdot P \tag{7}$$

K is used to encrypt any desired sensitive transaction data, e.g. δ, to get a ciphertext, C.

A **view tag**, t for this stealth address or transaction is the most significant byte of the hash of K:

$$t = \mathtt{keccak256}(K)[0:1] \tag{8}$$

t, x, Q and C are concatenated to form auxiliary data $(t \| x \| Q \| C)$ to be broadcasted along with the transaction.

A user parses the auxiliary data of a transaction and tries to calculate a shared key K' using its view private key p' and Q as:

$$K' = p' \cdot Q \tag{9}$$

And calculates a view tag v' from K' using equation 8. If $v' \neq v$, the user is not the receiver. It stops.

Otherwise, the receiver continues and uses K' to perform decryption of C to retrieve the plaintext data, e.g. δ', that was encrypted before. The receiver uses δ' to calculate the stealth address x' (using equation 6). The user performs an additional check that $x' = x$ (hence, $\delta' = \delta$) to be sure that it is indeed the receiver just in case the view tag match was a false positive.

To spend the asset the receiver is now able to prove a statement $\mathcal{S}_x$ defined as:

$$\mathcal{S}_x \equiv \text{Knowledge of } (s, \delta) \colon x = \mathtt{poseidon}(S, \delta) \tag{10}$$

The proof of $\mathcal{S}_x$ is included in the ZK proof $\boldsymbol{\pi}$ of the transaction to prove the ownership of x.

6.3 Core

Core smart contracts of the zkFi protocol. The core includes a **multi-transactional multi-asset pool**, meaning the pool supports multiple assets and can transact multiple assets in a single transaction. The on-chain ZK Verifier verifies the proof submitted during the transaction.

6.3.1 Setup. Let $\mathcal{T}$ be the Merkle tree whose leaves are calculated using the poseidon hash function (equation 25). The tree leaf nodes are subsequently filled with note **commitment**s in an append-only fashion.

A note is a tuple N of multiple elements:

$$N \equiv (e, x, v, \delta) \tag{11}$$

where, $e \in \mathbb{B}^{24}$ is the asset identifier and $x \in \mathbb{B}^{248}$ is the stealth address of the owner, $v \in \mathbb{B}^{248}$ is associated value of the note and δ is the blinding factor from which x was generated.

The **commitment**, c of a note is:

$$c = \mathtt{poseidon}(e, x, v) \tag{12}$$

Let σ be the signature generated with sign key s that authenticates the ownership of N as:

$$\sigma = \mathtt{Schnorr\_Sign}(c, s) \tag{13}$$

where `Schnorr_Sign` is as defined in 30.

Let η be the **nullifier** hash of note N defined as:

$$\eta = \mathtt{poseidon}(l, c, \delta) \tag{14}$$

where l is index of note commitment c as leaf node in $\mathcal{T}$.

Let's define a set of **public inputs**, ρ to the prover as:

$$\rho \equiv (R, \mathbf{V}, \mathbf{E}, \boldsymbol{\eta}^{in}, \mathbf{c}^{out})$$

where,

- R is root of $\mathcal{T}$.
- $\mathbf{V}$ is a list of m public values for each output note. A value V in list $\mathbf{V}$ is considered negative for any value leaving the pool, positive otherwise.
- $\mathbf{E}^{in}$ is a list of public asset identifiers corresponding to public values.
- $\boldsymbol{\eta}^{in} = (\eta_1, \eta_2, ..., \eta_n)$ is list of nullifier hashes corresponding to n input notes $\mathbf{N}^{in} = (N_1^{in}, N_2^{in}, ...N_n^{in})$.
- $\mathbf{c}^{out}$ is list of commitments of m output notes $\mathbf{N}^{out} = (N_1^{out}, N_2^{out}, ..., N_m^{out})$.

Similarly, we define a tuple of **private inputs** as:

$$\omega \equiv (\mathbf{e}^{in}, \mathbf{e}^{out}, \mathbf{v}^{in}, \mathbf{v}^{out}, \boldsymbol{\delta}^{in}, \mathbf{S}^{in}, \boldsymbol{\sigma}^{in}, \mathbf{x}^{out}, \mathbf{l}^{in}, \mathbf{o}^{in}) \tag{15}$$

where

- $\mathbf{e}^{in}$ and $\mathbf{e}^{out}$ are the lists of asset identifiers of input notes and output notes.
- $\mathbf{v}^{in} = (v_1^{in}, v_2^{in}, ..., v_n^{in})$ and $\mathbf{v}^{out} = (v_1^{out}, v_2^{out}, ..., v_m^{out})$ are the values of input notes, $\mathbf{N}^{in}$ and output notes in $\mathbf{N}^{out}$.
- $\boldsymbol{\delta}^{in}$ is a list of blinding factors of stealth addresses attached to input notes, $\mathbf{N}^{in}$.
- $\mathbf{S}^{in} = (S_1^{in}, S_2^{in}, ..., S_n^{in})$ are public keys associated with n input notes $\mathbf{N}^{in}$.
- $\boldsymbol{\sigma}^{in}$ are the signatures produced by signing commitments of $\mathbf{N}^{in}$ with private keys $\mathbf{s}^{in}$ corresponding to public keys $\mathbf{S}^{in}$.
- $\mathbf{x}^{out}$ are stealth addresses associated with output notes $\mathbf{N}^{out}$.
- $\mathbf{l}^{in}$ are leaf indices of input note commitments, $\mathbf{c}^{in}$.
- $\mathbf{o}^{in}$ are openings in $\mathcal{T}$ at indices $\mathbf{l}^{in}$ respectively.

Then, given ρ and ω, let $\mathcal{S}$ be the statement of knowledge defined as:

$$\begin{aligned}
\mathcal{S} \equiv \text{Knowledge of } \omega \colon & \\
& \forall i \in [1, n],\ x_i^{in} = \mathtt{poseidon}(S_i^{in}, \delta_i^{in}) \\
& \wedge \forall i \in [1, n],\ c_i^{in} = \mathtt{poseidon}(e_i^{in}, x_i^{in}, v_i^{in}) \\
& \wedge \forall i \in [1, n],\ \eta_i^{in} = \mathtt{poseidon}(l_i^{in}, c_i^{in}, \delta_i^{in}) \\
& \wedge \forall i \in [1, n],\ \mathtt{Schnorr\_Verify}(c_i^{in}, S_i^{in}, \sigma_i^{in}) = 1 \\
& \wedge \forall i \in [1, n],\ o_i^{in} \text{ is the opening at } l_i^{in} \text{ in } \mathcal{T} \\
& \wedge \forall i \in [1, m],\ c_i^{out} = \mathtt{poseidon}(e_i^{out}, x_i^{out}, v_i^{out}) \\
& \wedge \forall e_i^{in} \in \mathbf{e}^{in}, \sum \mathtt{selv}(\mathbf{v}^{in}, e_i^{in}) + \sum \mathtt{selv}(\mathbf{V}, e_i^{in}) \\
& \qquad\qquad = \sum \mathtt{selv}(\mathbf{v}^{out}, e_i^{in}) \\
& \wedge \forall\ e_i^{out} \in \mathtt{sele}(\mathbf{E}, \mathbf{e}^{out}), \sum \mathtt{selv}(\mathbf{v}^{in}, e_i^{out}) \\
& \qquad + \sum \mathtt{selv}(\mathbf{V}, e_i^{out}) = \sum \mathtt{selv}(\mathbf{v}^{out}, e_i^{out})
\end{aligned} \tag{16}$$

where `selv` is a function to filter select values from a list of values by a given asset identifier:

$$\mathtt{selv}(\mathbf{v}, e) = \{v \colon v \in \mathbf{v}\ \wedge v \text{ is the value of asset } e\} \tag{17}$$

and `sele` is a function to select elements from the first list parameter if non-zero, otherwise selecting from the second list parameter:

$$\texttt{sele}(\mathbf{E}, \mathbf{e}) = \{e \colon e = E_i \text{ if } E_i \neq 0, \text{ otherwise } e = e_i\} \quad (18)$$

`Schnorr_Verify` is defined in 31.

Let d_p and d_v be the proving and verifying keys created using a trusted setup ceremony. Let's define the SNARK proof generator function as:

$$\texttt{Prove} : \mathbb{B}^* \rightarrow \mathbb{B}^{2048}; (d_p, \rho, \omega) \mapsto \boldsymbol{\pi} \quad (19)$$

where $\boldsymbol{\pi}$ is called the **proof**.

And proof verifier as:

$$\texttt{Verify} : \mathbb{B}^* \rightarrow \{0, 1\}; (d_v, \boldsymbol{\pi}, \rho) \mapsto y \quad (20)$$

where y is a single bit for representing boolean $true$ or $false$ as a result of proof verification.

6.3.2 Creating Transaction.

A transaction request is constructed by following the steps below:

1. Identify the different assets $\mathbf{e} = (e_1, e_2, ...e_n)$ needed for transaction.
2. Scan the network and fetch encrypted notes data. Decrypt it using the viewing key and filter for notes, $N_i \in \mathbf{N}$ such that its asset id, $e \in \mathbf{e}$.
3. Choose a number of notes $\mathbf{N}^{in} \subset \mathbf{N}$, with commitment indices $\mathbf{l}_{in}$ in $\mathcal{T}$, to spend.
4. Compute nullifier hashes, η^{in} of notes $\mathbf{N}^{in}$.
5. Select a recent root, R of $\mathcal{T}$ and calculate tree openings $\mathbf{o}^{in}$ at indices $\mathbf{l}^{in}$.
6. Derive stealth addresses $\mathbf{x}^{out}$ of recipients and create output notes $\mathbf{N}^{out}$ associated with them. Also, calculate output commitments, $\mathbf{c}^{out}$.
7. Set $\mathbf{V}$ and $\mathbf{E}$ to control any public asset value, if any, going into or coming out of contracts.

6.3.3 Signing Transaction.

To authorize the transaction, signatures $\sigma^{\mathbf{in}}$ from the shielded account are required. Each signature $\sigma_i^{in} \in \sigma^{\mathbf{in}}$ is obtained by signing c_i^{in} of notes being spent ($\mathbf{N}^{in}$):

$$\sigma_i^{in} = \texttt{Schnorr\_Sign}(c_i^{in}, s) \quad (21)$$

6.3.4 Proving Transaction.

Together with signatures and parameters identified or calculated during creation, the public and private inputs i.e. ρ and ω are put together. Then proof $\boldsymbol{\pi}$ of the transaction generated:

$$\boldsymbol{\pi} = \texttt{Prove}(d_p, \rho, \omega) \quad (22)$$

6.3.5 Sending Transaction.

Except for the deposit transactions type (public value going into pool), the transaction is sent as an EIP-4337 user operation [12] by submitting it to a bundler node.

For deposits, it is sent directly from a wallet since it involves transferring value from the wallet to the pool smart contract.

If $\boldsymbol{\pi}$ was calculated correctly and the statement was true, the on-chain `Verify` function outputs 1, and flow proceeds with necessary funds. Otherwise, 0 representing bad or tampered $\boldsymbol{\pi}$.

6.4 Protocol Proxy

A **Protocol Proxy** is a state-less contract that lives on-chain to act as a proxy for a target (e.g. DeFi) protocol. This contract must implement a standard interface that exposes a "**convert**" functionality or operation. This reflects the fact that, in general, any DeFi operation e.g. swapping, lending, staking, can be thought of as a process of converting from one asset called *input asset* to another asset called *output asset*. Here are a few examples:

- When swapping ETH for USDC on Uniswap, ETH (input asset) is converted to USDC (output asset).
- When supplying DAI on the Aave market to earn interest, DAI (input asset) is essentially being converted to aDAI (output asset) the interest-bearing tokens given back in return.
- Staking ETH (input asset) on Lido results in conversion to stETH (output asset) holding which represents your stake plus rewards.

This allows **Core** the core to interact with protocols via a **Convertor** contract. In its **convert** operation, this proxy is supposed to perform necessary operations by calling the target it is the proxy for, eventually returning any output to the Core.

It is a specific kind of transaction where V values are set to non-zero so that these values, $\mathbf{v}$ are sent as input asset (with ids $\mathbf{e}$) if required. A fee asset of id e_f and value v_f is also given specifying gas fee requirements. Fee must be paid by returning it as one of the output assets. The target protocol may chose to pay gas however it wants. Lastly, the input also includes an arbitrary payload, d necessary for operation by the proxy. After processing the input, $(\mathbf{e}, \mathbf{v}, e_f, v_f, d)$ with its specific target protocol, the proxy is expected to return output asset $(\mathbf{e}', \mathbf{v}')$.

$$\begin{aligned}\texttt{Convert:}\ &(\mathbb{B}^{24^n}, \mathbb{B}^{256^n}, \mathbb{B}^{24}, \mathbb{B}^{256}, \mathbb{B}^*) \rightarrow (\mathbb{B}^{24^m}, \mathbb{B}^{256^m})\\ &; (\mathbf{e}, \mathbf{v}, e_f, v_f, d) \mapsto (\mathbf{e}', \mathbf{v}')\end{aligned} \quad (23)$$

Later, a note commitment is calculated with (e', v') for each of m outputs and a given stealth address as a transaction parameter and inserted into $\mathcal{T}$.

The flexibility comes from the fact that the implementation of `Convert` is left to the developer trying to integrate the target protocol.

6.5 Involuntary Selective De-anonymization

zkFi adopts a framework for involuntary de-anonymization of illicit transaction subgraphs using a threshold multi-party procedure while keeping such parties accountable.

The process involves an independent set of parties called guardians. Each of the n guardians holds a share of the secret key which decrypts encrypted transaction data such that guardians must reach a quorum of minimum size t ($t \leq n$) for the de-anonymization of any transaction to occur. However, even after de-anonymization is granted by guardians, they still do not learn any information about the user because another decryption still needs to be performed by an individual party called the revoker.

Accountability comes from the fact that the revoker is bound to send a publicly verifiable request of de-anonymization to the guardians who must agree first for the revoker to extract any information at all. A more detailed study can be found in [2].

7 Cryptographic Primitives

7.1 Hash Function

A hash function is a mathematical function, h that takes arbitrary length input, $m \in \mathbb{B}^*$, and outputs fixed-length output $x \in \mathbb{B}^n$ often called hash, hash value or digest.

$$h\colon \mathbb{B}^* \longrightarrow \mathbb{B}^n; m \longmapsto x \tag{24}$$

A cryptographically secure hash function has the following properties:

- **Pre-image resistance** Given a hash value, it must be difficult to find the input that produced it.
- **Second pre-image resistance** Given an input and its hash value it must be difficult to find another input that produces the same hash.
- **Collision resistance** It must be difficult to find two inputs that map to the same hash value.

Specifically for its ZK-related operations, zkFi extensively uses a hash function called **Poseidon** [9] denoted as `poseidon`:

$$\texttt{poseidon}\colon \mathbb{B}^* \longrightarrow \mathbb{B}^{254} \tag{25}$$

A common in the Ethereum ecosystem is the Keccak256 [10]. It is used within zkFi where it does not require proving via ZK proof. It is defined as:

$$\texttt{keccak256}\colon \mathbb{B}^* \longrightarrow \mathbb{B}^{256} \tag{26}$$

7.2 Elliptic Curve

Elliptic curves are mathematical structures that are defined over some finite field that have interesting properties that make them very useful in cryptography. The security of ECC relies on the computational infeasibility of solving the elliptic curve **discrete logarithm problem (DLP)** [19].

Let E be an elliptic curve defined over a finite field $\mathbb{F}_q$ where q is a large prime number, and let P, Q and R be points on E. The elliptic curve group operation on E is typically denoted as $P + Q$, where $P + Q + R$ satisfies the group law:

$$P + Q + R = O \tag{27}$$

where O represents the point at infinity.

The elliptic curve DLP is defined as finding an integer k such that $S = k \cdot P$, where $k \cdot P = P + P + ... + P$ (k times) is referred to as scaler multiplication of point P.

A cyclic subgroup is a subset of an elliptic curve group that is generated by a single point called **generator**, denoted by G. Elements of the subgroup are obtained when G is scalar multiplied by integers i.e. $k \cdot G$ where $k \in \mathbb{Z}$.

zkFi protocol uses a specific elliptic curve called **Baby JubJub** as defined in [11]. It is particularly well-suited for cryptography operations in zero-knowledge (ZK) applications.

7.3 Digital Signature

A digital signature σ is a verifiable piece of data produced by signing a message m with a private key s through some chosen signature scheme or function.

$$\texttt{Sign}\colon (\mathbb{B}^*, \mathbb{B}^k) \longrightarrow \mathbb{B}^n; (m, s) \longmapsto \sigma \tag{28}$$

The signature scheme can later verify that the signature σ was produced by an entity who knows the m as well as private key s:

$$\texttt{Verify}\colon (\mathbb{B}^n, \mathbb{B}^j) \longrightarrow \{0, 1\}; (\sigma, P) \longmapsto y \tag{29}$$

where P is public key corresponding to s and y is a single bit representing result of verification - $true$ or $false$.

A cryptographically secure signature scheme must have the following properties:

- **Authenticity** A valid signature implies that the message was deliberately signed by the signer only.
- **Unforgeability** Only the signer can produce a valid signature for the associated message.
- **Non-reusability** The signature of a message cannot be used on another message.
- **Non-repudiation** The signer of the message cannot deny having signed the message with a valid signature.
- **Integrity** Ensures that the contents of the message are not altered.

zkFi utilizes **Schnorr** [14] signature scheme for signing private transactions and verifying those signatures. The signing function is defined as:

$$\texttt{Schnorr\_Sign}\colon (\mathbb{B}^{256}, \mathbb{B}^{512}) \longrightarrow \mathbb{B}^{768}; (m, s) \longmapsto \sigma \tag{30}$$

and verification as:

$$\texttt{Schnorr\_Verify}\colon (\mathbb{B}^{256}, \mathbb{B}^{512}, \mathbb{B}^{512}) \longrightarrow \{0, 1\}; (m, S, \sigma) \longmapsto b \tag{31}$$

where b is a single bit representing result of verification $true$ or $false$.

7.4 Zero Knowledge Proofs

Zero-knowledge proof (ZKP) is a cryptographic technique that allows a party, a prover to prove to another party, a verifier that it knows a secret without actually revealing a secret by following a set of complex mathematical operations.

Although the origins of ZKPs can be traced back to the early 1980s, when a group of researchers, including Shafi Goldwasser, Silvio Micali, and Charles Rackoff, published a paper [20] that introduced the primitive concept to the world, it lacked practicality at the time. However, later developments addressed the problems leading to usable implementations in various systems, especially with the advent of blockchains.

In order to be ZKP, it must satisfy three properties:

- **Completeness**: If the statement is true, both the prover and verifier are following the protocol, then the verifier will accept the proof.
- **Soundness**: If the statement is false, no prover can convince the verifier that it is true with any significant probability.
- **Zero Knowledge**: If a statement is true, the verifier learns nothing other than the fact that the statement is true.

One particularly efficient type of ZKP is **zkSNARK**. A **SNARK** (Succinct Non-Interactive Argument of Knowledge) defines a practical proof system where the proof is **succinct** that can be verified in a short time and is of small size. The system is **non-interactive** so that the prover and verifier do not have to interact over multiple rounds. **Knowledge** refers to the fact that the statement is true and the prover knows a secret, also called "**witness**" that establishes that fact. If a SNARK proof allows for proof verification without revealing a witness it becomes a zkSNARK.

Generating a zkSNARK proof is a multi-step process that involves breaking down logic to primitive operations like addition and multiplication, creating an **Arithmetic Circuit** consisting of gates and wires. This circuit is then converted to a **R1CS** (Rank-1 Constraint System) which constrains the circuit to verify, for instance, that values are being calculated correctly with gates. The next step converts these constraints in the circuit to a Quadratic Arithmetic Program (**QAP**). QAP represents these constraints with polynomials rather than numbers. Afterward, together with Elliptic Curve Cryptographic (ECC) and QAP, a zkSNARK proof is formed.

7.4.1 Trusted Setup Ceremony.

A trusted setup ceremony is a cryptographic procedure that involves contributions of one or more secret values, s_i from one or more parties into the trusted setup system to eventually generate some piece of data that can be used to run cryptographic protocols. Once this data is generated, the secrets, s_i are considered **toxic waste** and must be destroyed because possession of these makes it possible to rig the system. In the context of ZK, for example, it can be used to generate proofs that are false positives.

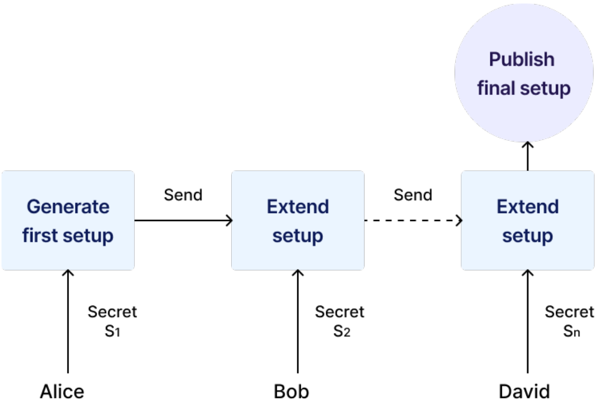

Figure 3. Trusted Setup Ceremony

There are multiple kinds of trusted setup procedures. The original ZCash ceremony in 2016 [16] was one earliest trusted setups being used in a major protocol. One trusted setup of particular interest is **powers-of-tau** setup which is multi-participant and is commonly used in protocols. To get an idea of how multi-step setups work consider the image below where, secrets, $s_1, s_2, ..., s_n$

To be a bit precise, the secrets are used in Elliptic Curve operations for security. An input s_i is used to generate successive points on the curve given the generator G by operation $G * s_i$. The result of the contribution, $G * s_i$ is fed to the next step. The next participant cannot determine the previous contributor's secret because it is a discrete log.

A more detailed discussion can be found at [17].

7.4.2 Groth16.

Groth16 is a SNARK proving system proposed by Jens Groth [18] in 2016. It is a non-interactive proof system that is commonly used in privacy protocols and is fast with small proofs. The prover complexity is $O(n_c)$ where n_c is the number of rank-1 constraints.

Groth16 requires a trusted setup to generate proving keys and verifying keys. The setup is required to make the proofs succinct and non-interactive.

7.5 Merkle Tree

Merkle Tree is a data structure that is used to store and verify the integrity of data. It is a binary tree where each node has two children. The leaves of the tree are the data blocks that are stored and the internal nodes of the tree are the hashes of children below them. Consequently, the root of the tree is the culmination of all data blocks in the tree. This means that if any data blocks at the leaf nodes are changed or corrupted it will propagate to the root which will change too.

The root hash of the Merkle tree can be compared to a reference to verify the integrity of the data. This makes the Merkle trees very efficient.

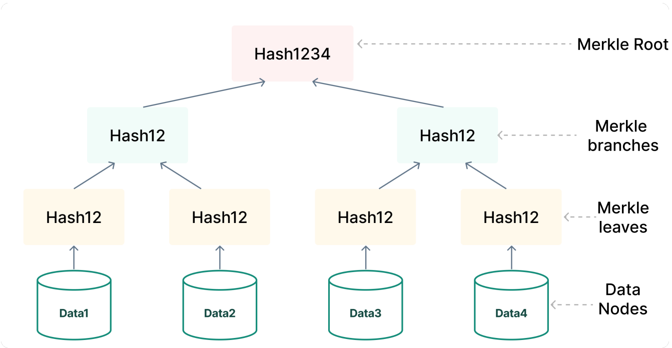

Figure 4. Merkle tree

A merkle tree is maintained in zkFi to record user assets. User-deposited assets are represented in the form of a hash called **commitment** that gets stored in a Merkle tree structure. During withdrawal, the same user submits a proof a.k.a **Merkle Proof** (among others) of inclusion of the same commitment hash in the tree and proof of knowledge of the hash's pre-image, without revealing any traceable information like an index of commitment in tree or constituents of pre-image.

8 Conclusion

For web3 to reach mass adoption, privacy is an unquestionable facet. While zero-knowledge technology has solved the privacy problem, earlier solutions have clearly indicated that strong compliance to prevent illicit use cannot be ignored at all. Furthermore, to encourage protocols and developers to build privacy into their products, infrastructure solutions must be present that are easy to integrate and build on top of.

In this paper, we demonstrated that an infrastructure-based privacy solution with built-in compliance is possible. We proposed a middleware solution through an SDK package that is easy to integrate and abstracts away solutions to privacy and compliance without sacrificing developer experience. Consequently, rendering the protocols and developers the freedom to innovate and focus on solving their core problem instead of worrying about user privacy or compliance.

References

[1] Joseph Burleson, Michele Korver, and Dan Boneh. *Privacy-Protecting Regulatory Solutions Using Zero-Knowledge Proofs*. Available at https://api.a16zcrypto.com/wp-content/uploads/2022/11/ZKPs-and-Regulatory-Compliant-Privacy.pdf

[2] Amit Chaudhary and Hamish Ivy-Law. *Balancing Blockchain Privacy and Regulatory Compliance By Selective De-Anonymization*. https://labyrinth.ac/Sede_privacy_compliance_v3.pdf

[3] Etherscan. (2023). https://etherscan.io/

[4] Metamask Snaps. (2023). https://metamask.io/snaps/

[5] Ledger Developer Portal. (2023). https://developers.ledger.com/docs/embedded-app/introduction/

[6] Uniswap Docs. (2023). https://docs.uniswap.org/concepts/protocol/oracle

[7] TRM Labs, Inc. *U.S. Treasury Sanctions Widely Used Crypto Mixer Tornado Cash* https://www.trmlabs.com/post/u-s-treasury-sanctions-widely-used-crypto-mixer-tornado-cash

[8] Chainalysis Team *Crypto Mixer Usage Reaches All-time Highs in 2022, With Nation State Actors and Cybercriminals Contributing Significant Volume* https://blog.chainalysis.com/reports/crypto-mixer-criminal-volume-2022/

[9] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. *POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems* https://eprint.iacr.org/2019/458.pdf

[10] NIST (August 2015). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[11] ERC-2494: Baby Jubjub Elliptic Curve. https://eips.ethereum.org/EIPS/eip-2494

[12] ERC-4337: Account Abstraction Using Alt Mempool. https://eips.ethereum.org/EIPS/eip-4337

[13] ERC-5564: Stealth Addresses. https://eips.ethereum.org/EIPS/eip-5564

[14] C.P. Schnorr. *Efficient Signature Generation by Smart Cards.* https://d-nb.info/1156214580/34

[15] Nir Yoav, Langley Adam. *ChaCha20 and Poly1305 for IETF Protocols.* https://datatracker.ietf.org/doc/html/rfc8439

[16] Morgan E. Peck. *The Crazy Security Behind the Birth of Zcash, the Inside Story.* https://spectrum.ieee.org/the-crazy-security-behind-the-birth-of-zcash

[17] Vitalik Buterin. *How do trusted setups work?.* https://vitalik.ca/general/2022/03/14/trustedsetup.html

[18] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments.* https://eprint.iacr.org/2016/260.pdf

[19] N. Koblitz. *A Course in Number Theory and Cryptography, Second Edition* (1994), 97-107.

[20] Shafi Goldwasser, Silvio Micali, and Charles Racko. *The Knowledge Complexity Of Interactive Proof Systems.* https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Proof%20Systems/The_Knowledge_Complexity_Of_Interactive_Proof_Systems.pdf