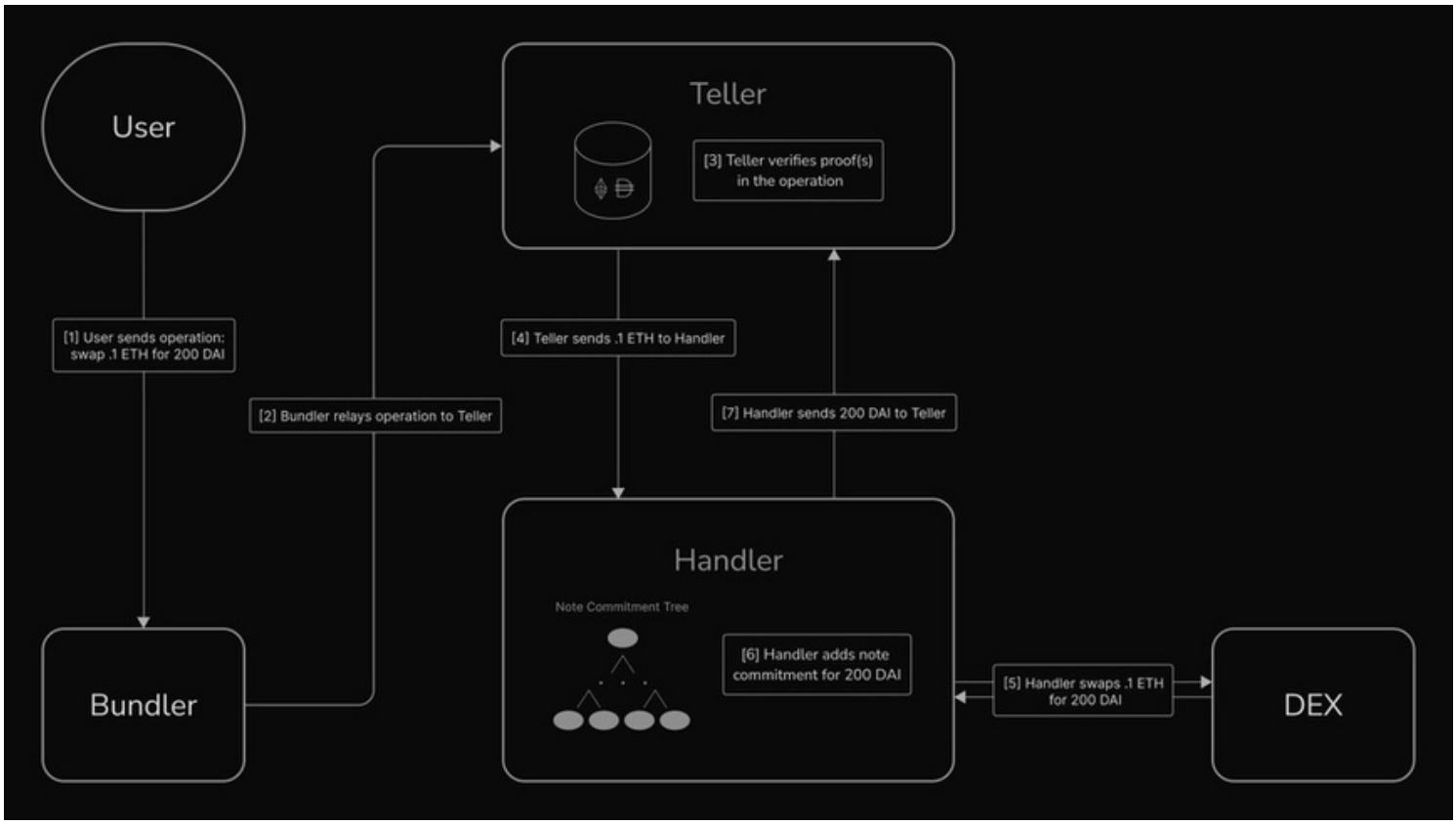
THE SPECTER PROTOCOL

LITEPAPER v1

Introduction

What is Specter?

Specter is a protocol that enables private accounts on Ethereum. Imagine a conventional Ethereum account but with built-in asset privacy. Specter allows users to deposit or receive funds to private, stealth addresses within the Specter contracts. Then, in the future, a user can prove ownership of assets in zero knowledge for use in arbitrary transactions or confidential transfers.



How can I use Specter?

Specter will initially be available via the private vault UI upon mainnet launch. This interface will allow users to privately store/earn yield on idle assets and withdraw to clean burner wallets for higher-touch activities like trading. This is intended to be the home for users' hot assets which they know they will be transacting with but will want a fresh wallet to do so with each time.

Protocol Overview

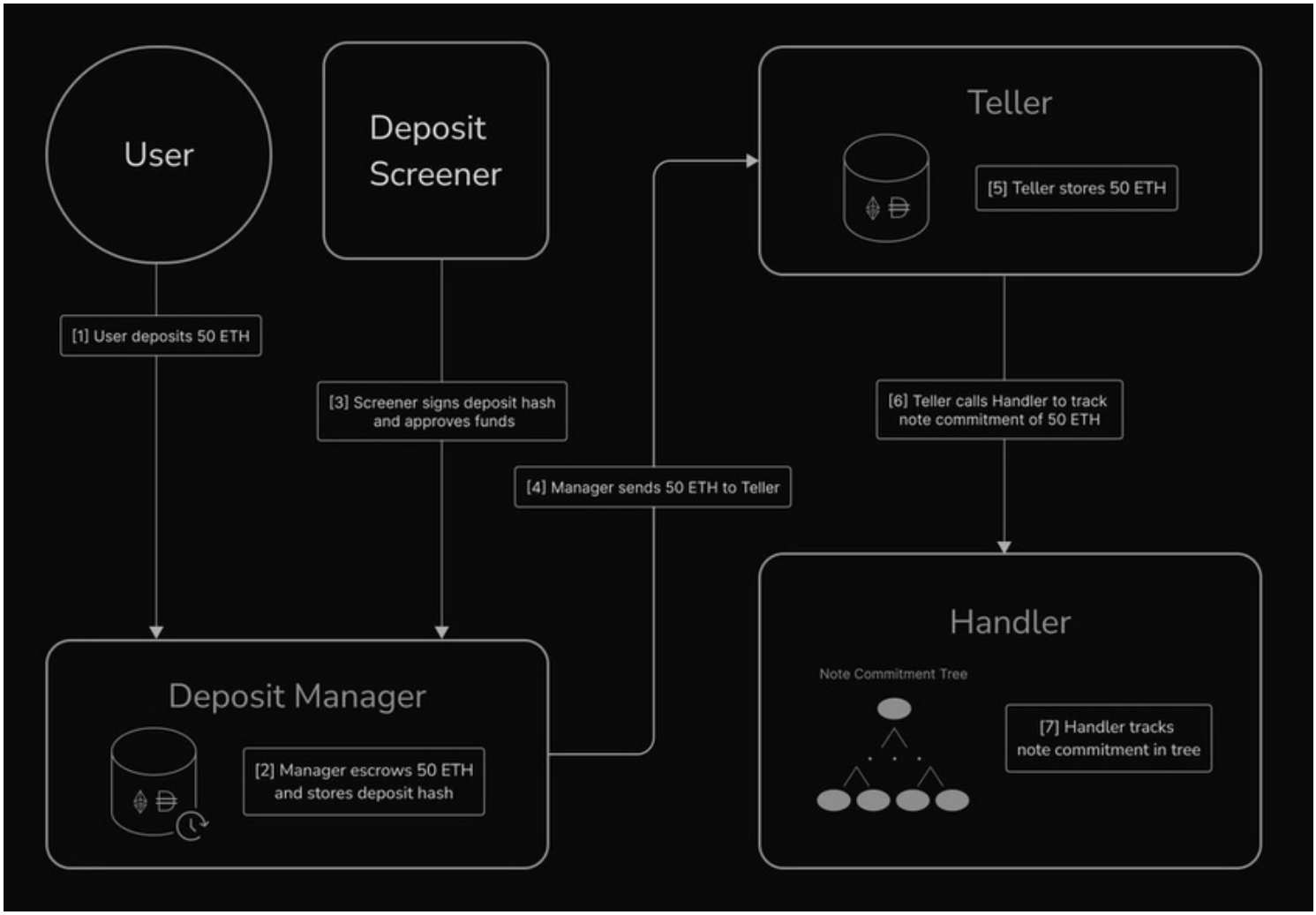
High Level

Specter is a private account abstraction protocol.

At a high level, users can deposit assets into the protocol to one of their stealth addresses. Later, in the future, they can prove ownership of said assets in zero-knowledge for use in arbitrary anonymous contract interactions or confidential payments.

Deposits

Deposits are how a users can move assets into Specter such that they can be transacted with privately in the future. Currently, in order to minimize the inflow of illicit funds, deposits into Specter initially go into the Deposit Manager contract. Assets will wait in the contract until an offchain actor called the screener signs off on and completes deposits below a certain compliance risk threshold. Please see our for the rationale behind the design decision and our long term plans for improving permissionlessness.



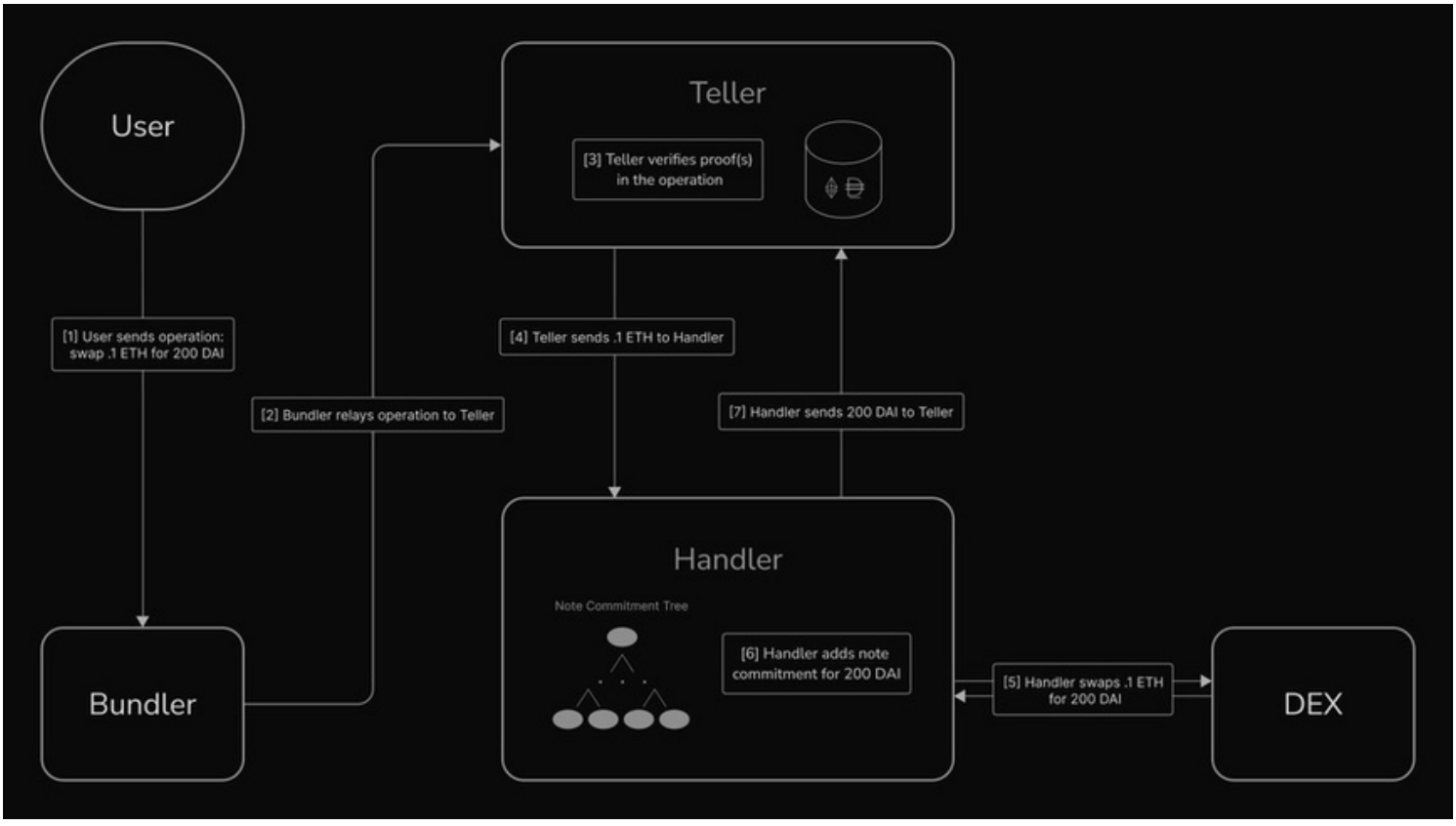
Operations

Once funds have been deposited, all usage of private funds are initiated from a *single* Teller contract, which takes a bundle of operations, verifies their proofs, then delegates the processing and execution of individual operations to a second contract called the Handler. To initiate a dapp interaction, a user constructs an Operation, which encodes some assets to spend and a sequence of contract calls. Specter guarantees that:

1- Right before contract calls are executed, the Handler contract holds *exactly* the assets declared in Operation.
2- Contract calls are carried out directly from the Handler contract.

3- After the calls are completed, all leftover tokens in the Handler contract are "refunded" to the initiating user's stealth address.

To make sure each user Operation only spends the assets that the user has declared, deposited assets are held by the Teller, separate from where execution occurs. The Handler contract will request only those assets declared by the Operation, then perform the contract calls. Once the contract calls are made, the Handler will send any remaining or new funds back to the Teller. The Handler will then create new note commitments for these assets belonging to the user. See the following diagram for a high level view of the workflow.



Keys and Stealth Addresses

The core mechanism that allows generalized contract calls through Specter is a scheme for re-randomizable addresses, which we call ***stealth addresses***.

The main idea is to separate the ability to "view" transactions and the ability to "sign" transactions into two separate "private keys" - a ***viewing key*** and a ***spending key***.
To avoid complicating key management, the viewing key can be derived from the spending key, but the spending key cannot be derived from the viewing key. In other words:
1- Someone with access to only a user's viewing key can trace the user's transactions, but cannot spend the user's funds.
2- Someone with access to the user's spending key can spend the user's funds *and* trace their transactions
3- Someone without access to either key cannot trace the user's transactions or spend their funds.

From the viewing key, we then derive an associated address, which we refer to as the ***canonical address***. A user's canonical address is analogous to a user's Ethereum address - it serves as a pseudonym that uniquely identifies an account.

However, unlike an Ethereum address, any user can ***randomize*** any other user's canonical address into a one-time ***stealth address*** without access to the user's viewing key. This stealth address refers to the same account as the canonical address, however, third parties cannot determine the association without access to the corresponding viewing key (i.e. it "looks random") . In the same manner, any user can also ***re-randomize*** any stealth address and generate new stealth addresses.
Consider a user Alice. She can generate any number of one-time stealth addresses that can only be linked via their viewing key.
Let's say Alice has a friend named Bob. She trusts Bob, so Alice shares her canonical address with him. He can generate stealth addresses to pay Alice without revealing her canonical address or knowing her viewing key.

Now consider a third user named Charlie who's a stranger to Alice, so Alice doesn't want to give him her canonical address. Instead, Alice can give him one of her stealth addresses, or he can get one from Bob. Then, if Charlie wants to pay Alice, he can re-randomize the stealth address and pay Alice using the re-randomized address.

Notes, Commitment Tree, Nullifiers, and JoinSplits

Just like ZCash Sprout, Specter performs all bookkeeping through a UTXO system of *notes* stored in a commitment tree. Notes are spent via *JoinSplits* and double-spends of notes prevented via *nullifiers*. Before we dive further into the protocol, we must first understand these concepts.

Notes

A ***note*** can conceptually be thought of as a "dollar bill" with an owner. It takes the form of a record with three fields:
1- owner: an anonymous stealth address for the note's owner (we will cover these in the next section)
2- asset: a field indicating the asset the note is for. Currently, Specter has the capability to support ERC20 (fungible tokens), ERC721 (NFTs), and ERC1155 (semi-fungible tokens).
3- value: a number indicating how much of the asset this note is for.

A ***note commitment*** is the cryptographic hash of the note. That is, H(note).

Notes can be acquired in one of three ways:

1- By depositing funds into the Specter contracts (i.e. "wrapping" them)

2- By receiving a "refund" containing all of the unspent assets left over at the end of an "operation"
3- By receiving a note through a JoinSplit (i.e. a "confidential payment")

Commitment Tree

On-chain, the Handler contract maintains a Merkle tree of the note commitments for all "valid" notes - that is, notes that were created by one of the above three events (deposits, refunds, and confidential payments). We refer to this tree as the ***commitment tree***. In the process of spending a note, one must prove that the note commitment is in the commitment tree and is thus a valid UTXO in the system.

Nullifiers

To spend a note, one needs the owner's viewing key *and* spending key, the spending key is used to sign the transaction, and the viewing key is used to derive a secret number called a ***nullifier.***
Every note has one nullifier, and every nullifier corresponds to one note. Nullifiers can only be derived using the owner's viewing key.
On-chain, the Handler contract keeps track of a set of nullifiers. Each time a note is spent, its nullifier is revealed and the contract will check if the revealed nullifier is in the set:
1- If it's not in the set, the contract will accept the note, proceed with the transaction, and add the nullifier to the set.
2- If it *is* in the set, then the contract will reject the note and revert.

Since each nullifier is uniquely tied to a note, nullifiers prevent double-spends. The set of all spendable notes in the tree is the set of all notes in the note commitment tree that have not yet been nullified.

JoinSplits

All Notes are spent via an operation called a ***JoinSplit***. A ***JoinSplit*** is an operation through which two *input notes* are spent by revealing their nullifiers, and two *output notes* are created. The notes' owner must sign this operation with their spending key. They can choose the owners and amounts of the output notes, subject to the following constraints:
1- The total value of the input notes equals the total value of the output notes plus an additionally-declared ***public spend amount***, which may be 0.
2- The input notes and output notes all have the same asset

3- Both notes' commitments are in the commitment tree

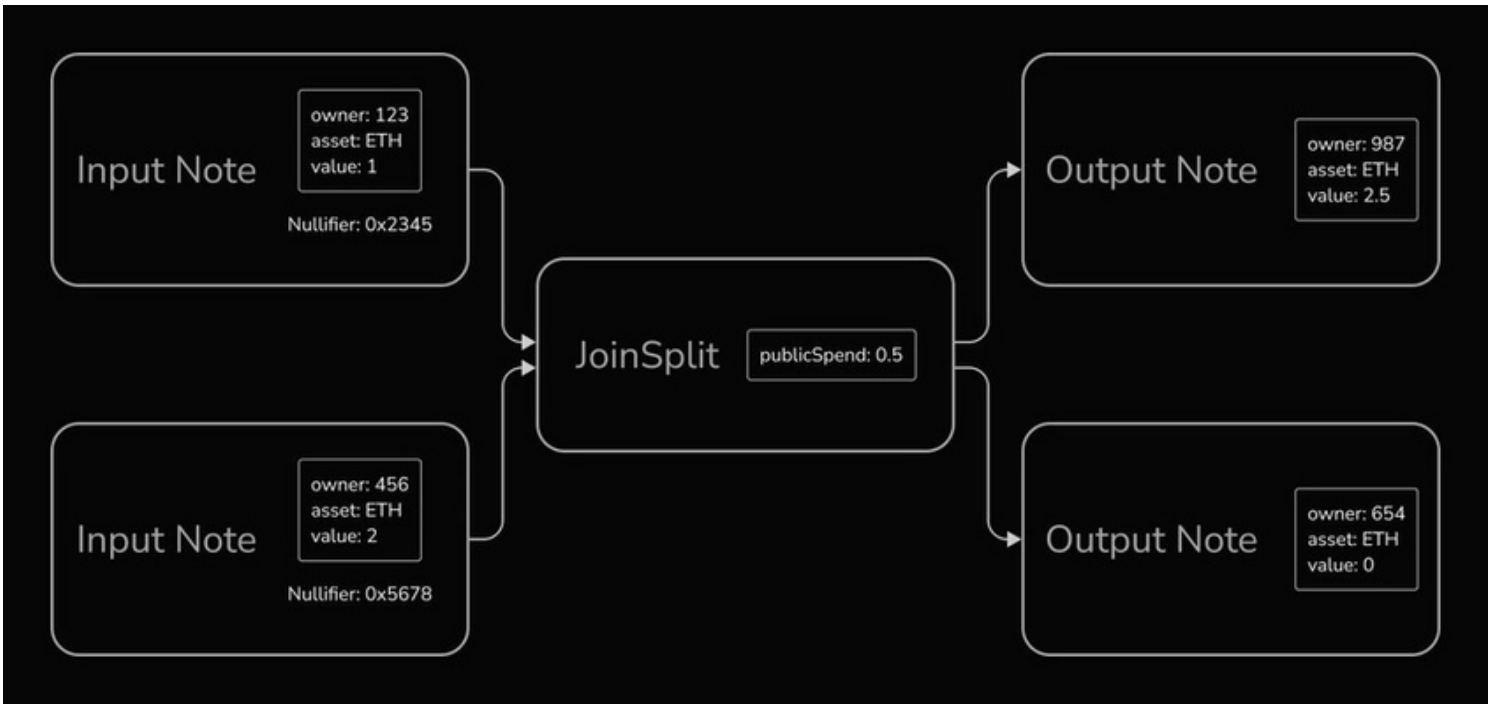
(If the second input note's value is 0, the merkle proof for it is ignored by the JoinSplit circuit. This makes it so that the user isn't forced to spend two notes.)
Only the following information is revealed on-chain:

1- the asset to spend, unless the public spend amount is 0

2- the public spend amount (can be 0)

3- the nullifiers for both input notes

4- the commitments for both output notes



By choosing the values/owners of the output notes, we can use a JoinSplit to perform a wide variety of tasks:

1- We can "merge" together small notes by one of the output note values to 0 and the other to the total input value. Going with the "dollar bill" analogy, this is like combining 2 $1 bills into a single $2 bill.

2- We can pay another user within the protocol without revealing the amount by setting the owner to another user's stealth address. We call this a "confidential payment"
3- We can unwrap assets from within the pool for usage in an operation by setting the total output amount to be less than the total input amount and specifying a nonzero public spend amount.

4- We can do a combination of the above - for instance, in a single JoinSplit, I can unwrap some ETH for a Uniswap transfer AND send a confidential ETH payment to a friend.

Deposits

To make a deposit, the user first *instantiates* a deposit by calling a contract called the *Deposit Manager*. Instantiating a deposit entails two things:

1- Specifying the asset/amount the user would like to deposit + the stealth address they would like to "own" the deposited funds
2- Sending the specified asset and amount to the Deposit Manager contract

The Deposit Manager is a temporary "waiting room" where pending deposit request and related funds sit. If the user would like to retrieve their deposit, they can call on the Deposit Manager contract to retrieve their funds at any time.

While the funds sit in the Deposit Manager, a permissioned actor called the deposit screener polls for new deposits submitted to the Deposit Manager, signs all deposits under an acceptable risk threshold, and "completes" the deposit by submitting their signature to the Deposit Manager.

Upon receipt of a "complete deposit" transaction containing a valid screener signature, the Deposit Manager will move the funds into the primary "entry point" of the protocol - the Teller. Additionally, the deposit completion will trigger a third contract called the *Handler* to list the deposited asset as the asset, the deposited amount as the amount, and the deposit address as the note's owner.

Once the note is created, it is queued for insertion into an on-chain merkle tree called the *commitment tree*. The commitment tree is a merkle tree of all notes ever created by the protocol. Later, when the user wishes to spend the note, they will prove its validity inside a ZKP by proving membership of the note in the tree.
Once the note is queued for insertion, the transaction is complete. The note will soon be inserted in a batch together with other new notes to save gas.

Operations

Within the Specter protocol, users carry out actions with their private funds through *operations*. Abstractly, an operation consists of two pieces of information: A list of Joinsplits and a list of *actions*.

An *action* consists of a contract address and some ABI-encoded calldata specifying a method call to make on that contract.
At a high level, an easy way to interpret an operation is:

1- The list of JoinSplits specifies what assets to spend and how much of it to use for actions, all without revealing the owner of the assets

2- The list of actions specifies what to *do* with said assets

Concretely operations contain other important fields for security, including:

1- A stealth address refundAddr that specifies an owner for any new notes created at the end of the operation

2- Gas price / limit

3- Chain id / deadline to protect against replay attacks

Once submitted, operations are processed on-chain with the following procedure:

1- The JoinSplit proofs are verified by the Teller against the committment tree.

2- The old note from the JoinSplits are checked against the nullifier set in the Handler to prevent double spends.
3- The old note nullifiers from the JoinSplits are added to the nullifier set.

4- The new note commitments produced by the JoinSplits are queued for insertion into the commitment tree.
5- Any publicSpend in the JoinSplits is sent from the Teller to the Handler (unwrapped).

6- The actions are executed by the Handler using the unwrapped funds.

7- Any new refund notes resulting from the actions are queued for insertion into the commitment tree.
8- The underlying assets for the refund notes are sent back to the Teller contract.

To better understand what this means, let's consider some practical examples.

Confidential Payment

A "confidential payment" is a payment where the amount and recipient are hidden. We can represent a confidential payment via an operation with no actions and one or more JoinSplits where there is a publicSpend of 0 and the owner of one of the new notes is set to the payment recipient.

Since publicSpend equals 0, the underlying assets are never unwrapped and thus are never revealed, making the payment fully confidential. All that is seen on chain is that some unknown notes for unknown assets and amounts were spent and new unknown notes of that same unknown asset were created.

Anonymous ERC-20 Transfer

Suppose we have some USDC inside Specter and we wanted to unwrap it and pay an Ethereum address $100. In this interaction, the sender is anonymous but the recipient is not.
We can express this transfer via an operation with one or more JoinSplits for USDC, where the public JoinSplits' spends total to $100. The operation would have one action encoding a call to transfer $100 USDC to the recipient.

When submitted, this operation will unwrap the desired amount of USDC from Specter, then transfer it to the recipient.

Anonymous Uniswap Call

Suppose we have 3 WETH inside Specter and we wanted to swap 1 WETH for DAI. We can create an operation with one or more JoinSplits for WETH, where the JoinSplits' public spends total to 1 WETH. The operation would have one action encoding a call to Uniswap to swap 1 WETH for DAI.
When submitted, this operation will unwrap 1 WETH from Specter, call Uniswap to swap it for DAI, create a refund note for the resultant DAI, and send the DAI back to the Teller.