SECURITY ANALYSIS OF PHOENIX VERSION 1.0

DUSK

Abstract. This document contains a formal security analysis of the Phoenix protocol. It contains a brief description of the protocol itself, although the document is not self-contained, and is intended to be used in conjunction with the Phoenix documentation. It also contains security models for non-malleability, ledger indistinguishability, balance and note spendability, along with proofs that Phoenix satisfies each of these properties.

Contents

Date: April 16, 2024.

1. Notation

We use $\mathsf{sansserif}$ font for algorithms and objects. Algorithms are uppercase, whereas objects are lowercase. We use **bold** font for oracle queries.

We often simplify subscripts and superscripts when they are the same for all elements of a tuple. For example, we write $(a, b, c)_i^j$ as shorthand for (a_i^j, b_i^j, c_i^j).

- $\perp$: a symbol representing that an algorithm terminated with an error.
- $\triangle$: a placeholder value, used when a field is not needed. Its concrete choice is an implementation detail.
- $a \mid\mid b$: concatenation of two elements a, b.
- $x \in S$ or $x \notin S$: the element x belongs / does not belong to the set S.
- $x \leftarrow S$: the element x is sampled uniformly at random from the set S.
- S^n: set of n-tuples formed by elements of S.
- $\mathbb{F}_n$: finite field with n elements.

2. DAPs and security definitions

The security notions and proofs are inspired by those of Zerocash [BSCG+14], with influence from [GH19, Hop22]. They have been modified to the particularities of the Phoenix protocol.

Throughout this section, let

$$\Pi = (\mathsf{Setup}, \mathsf{CreateKeys}, \mathsf{Mint}, \mathsf{PrepareTransfer}, \mathsf{ProveTransfer}, \mathsf{VerifyTransaction}, \mathsf{Scan})$$

be a DAP scheme.

2.1. **Oracle queries.** A *DAP oracle* $\mathcal{O}$ is initialized with public parameters pp and is stateful. $\mathcal{O}$ stores the following data structures:

- ledger: an append-only list of transactions, which can be of type mint or $\mathsf{transfer}$.
- $\mathsf{honestKeys}$: a list of key tuples $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$ corresponding to honest users.
  - $\mathsf{scanKeys}$: a sublist of $\mathsf{honestKeys}$ that corresponds to keys that have enlisted the adversary as a helper to scan the ledger on their behalf.
  - $\mathsf{independentKeys}$: a sublist of $\mathsf{honestKeys}$ that contains the key tuples that are not in $\mathsf{scanKeys}$.
- $\mathsf{noteList}$: a list of the contents of notes.
- $\mathsf{honestTransfers}$: a list of transaction skeletons corresponding to transactions of type $\mathsf{transfer}$ that occur in response to **PrepareTransfer** or **FullTransfer** queries. Note that $\mathsf{honestTransfers}$ does not include those transfers that come from **Insert** queries.

We denote by MT the Merkle tree of note commitments in ledger. A transaction tx contains notes of the form $\mathsf{note} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)$. Whenever a transaction is added to ledger, each note it contains is associated to an empty leaf of MT, and thus associated a position pos. Leaves are never overwritten.

Given a query Q, the oracle answers differently, depending on the type of query. If, in any query type, an operation fails, output $\perp$, and in that case all the data structures remain unchanged.

For lists of keys, we often abuse notation and write e.g. $\mathsf{pk} \in \mathsf{honestKeys}$ as shorthand for "there exists an entry in $\mathsf{honestKeys}$ with public key pk".

Figure 1 contains the types of queries that a DAP oracle can receive and answer.

- $Q = (\textbf{CreateKeys}, \mathsf{scanHelp})$:
  1. Compute $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk}) = \mathsf{CreateKeys}(\mathsf{pp})$.
  2. Add $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$ to $\mathsf{honestKeys}$.
  3. If $\mathsf{scanHelp} = \mathsf{true}$:
     - Add $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$ to $\mathsf{scanKeys}$.
     - Output $(\mathsf{pk}, \mathsf{vk})$.

     If $\mathsf{scanHelp} = \mathsf{false}$:
     - Add $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$ to $\mathsf{independentKeys}$.
     - Output pk.
- $Q = (\textbf{Mint}, v, \mathsf{pk})$:
  1. Check that $\mathsf{pk} \in \mathsf{honestKeys}$.
  2. Compute $(\mathsf{noteContent}, \mathsf{note}) = \mathsf{Mint}(\mathsf{pp}, v, \mathsf{pk})$.
  3. Add $\mathsf{noteContent}$ to $\mathsf{noteList}$.
  4. Add $\mathsf{tx}^{\mathsf{mint}} = (\mathsf{note}, v)$ to ledger.
  5. No output.
- $Q = (\textbf{PrepareTransfer}, \{(\mathsf{pos}, \mathsf{pk})_i^{\mathsf{old}}, (v, \mathsf{pk})_i^{\mathsf{new}}\}_{i=1,2}, (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}}))$:
  1. Compute root of T_L.
  2. For $i = 1, 2$:
     (a) Let $\mathsf{com}_i^{\mathsf{old}}$ be the commitment at leave $\mathsf{pos}_i^{\mathsf{old}}$ in T_{ledger}.
     (b) Let tx_i be the mint/transfer transaction in ledger that contains $\mathsf{com}_i^{\mathsf{old}}$.
     (c) Let $\mathsf{noteContent}_i^{\mathsf{old}}$ be the first note in $\mathsf{noteList}$ with note commitment $\mathsf{com}_i^{\mathsf{old}}$.
     (d) Let $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})_i^{\mathsf{old}}$ be the first key tuple in $\mathsf{honestKeys}$ with $\mathsf{pk}_i^{\mathsf{old}}$ being $\mathsf{noteContent}_i^{\mathsf{old}}$'s associated public key.
     (e) Compute a Merkle proof $\mathsf{path}_i^{\mathsf{old}}$ from $\mathsf{com}_i^{\mathsf{old}}$ to root.
  3. Compute
  $$(\{\mathsf{noteContent}_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{tx}_{\mathsf{skeleton}}, \mathsf{publicInputs}, \mathsf{secretInputs}) = \\ = \mathsf{PrepareTransfer}\left(\begin{array}{l} \mathsf{pp}, \mathsf{root}, \mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}}, \\ \{(\mathsf{noteContent}, \mathsf{sk}, \mathsf{pos}, \mathsf{path})_i^{\mathsf{old}}, (v, \mathsf{pk})_i^{\mathsf{new}}\}_{i=1,2} \end{array} \right).$$
  4. Compute
  $$\mathsf{proof} = \mathsf{ProveTransfer}(\mathsf{pp}, \mathsf{publicInputs}, \mathsf{secretInputs}).$$
  5. Check $\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}^{\mathsf{transfer}}, \mathsf{ledger}) = \mathsf{accept}$.
  6. Add $\mathsf{tx}_{\mathsf{skeleton}}$ to $\mathsf{honestTransfers}$.
  7. Output $(\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{publicInputs}, \mathsf{secretInputs})$.
- $Q = \big(\textbf{FullTransfer}, \{(\mathsf{pos}, \mathsf{pk})_i^{\mathsf{old}}, (v, \mathsf{pk})_i^{\mathsf{new}}\}_{i=1,2}, (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{gasSpent}, \mathsf{pk}_{\mathsf{change}})\big)$:
  1. Run the query
  $$(\textbf{PrepareTransfer}, \{(\mathsf{pos}, \mathsf{pk})_i^{\mathsf{old}}, (v, \mathsf{pk})_i^{\mathsf{new}}\}_{i=1,2}, (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}})),$$
  obtaining $(\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{publicInputs}, \mathsf{secretInputs})$.
  2. Compute
  $$\mathsf{proof} = \mathsf{ProveTransfer}(\mathsf{pp}, \mathsf{publicInputs}, \mathsf{secretInputs}),$$
  3. If $\mathsf{gasSpent} > \mathsf{gasLimit}$, output $\perp$, else set $v = (\mathsf{gasLimit} - \mathsf{gasSpent}) \cdot \mathsf{gasPrice}$.
  4. Set $\mathsf{note}_{\mathsf{change}} = (v, \triangle, \mathsf{npk}_{\mathsf{change}}, R_{\mathsf{change}})$.
  5. Set $\mathsf{tx}^{\mathsf{transfer}} = (\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent})$.
  6. Check $\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}^{\mathsf{transfer}}, \mathsf{ledger}) = \mathsf{accept}$.
  7. Add $\mathsf{noteContent}_i^{\mathsf{new}}$, for $i = 1, 2$, and $\mathsf{note}_{\mathsf{change}}$ to $\mathsf{noteList}$.
  8. Add $\mathsf{tx}_{\mathsf{skeleton}}$ to $\mathsf{honestTransfers}$.
  9. Add $\mathsf{tx}^{\mathsf{transfer}}$ to ledger.
  10. No output.
- $Q = (\textbf{Receive}, \mathsf{pk})$:
  1. Let $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$ be the first tuple in $\mathsf{honestKeys}$ with public key the given pk.
  2. Compute $\{\mathsf{noteContent}_i\}_{i=1}^n = \mathsf{Scan}(\mathsf{pp}, (\mathsf{vk}, \mathsf{pk}), \mathsf{ledger})$.
  3. For $i = 1, \ldots, n$:
     (a) Add $\mathsf{noteContent}_i$ to $\mathsf{noteList}$.
     (b) Parse $\mathsf{noteContent}_i = (v, s, \mathsf{pk}, \mathsf{com}, R)_i$
     (c) Output com_i.
- $Q = (\textbf{Insert}, \mathsf{tx})$:
  1. Check $\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}, L) = \mathsf{accept}$.
  2. Add tx to ledger.
  3. For all pk in $\mathsf{honestKeys}$, run the query $(\textbf{Receive}, \mathsf{pk})$, updating $\mathsf{noteList}$ in the process.
  4. No output.

FIGURE 1. Types of oracle queries.

2.2. **Non-malleability.**

Definition 1. *We say that* Π *is* NM*-secure if for any PPT adversary* $\mathcal{A}$*, we have that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{NM}}(\mathcal{A}) < \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{NM}}(\mathcal{A})$ *is the advantage of* $\mathcal{A}$ *in the game described below.*

The non-malleability game NM*.*

- *Setup phase.* The challenger $\mathcal{C}$ runs $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$, sends pp to $\mathcal{A}$. $\mathcal{C}$ initializes a DAP oracle $\mathcal{O}$, with its corresponding ledger.
- *Query phase.* $\mathcal{A}$ submits queries Q to elicit behavior on honest users. $\mathcal{C}$ forwards Q to $\mathcal{O}$, and relays the answer back to $\mathcal{A}$. There is no special restriction on these queries.
- *Answer phase.* $\mathcal{A}$ outputs a transaction $\mathsf{tx}^* = (\mathsf{tx}^*_{\mathsf{skeleton}}, \mathsf{proof}^*, \mathsf{note}^*_{\mathsf{change}}, \mathsf{gasSpent}^*)$ of type $\mathsf{transfer}$. $\mathcal{A}$ wins iff
  (1) $\mathsf{tx}^*_{\mathsf{skeleton}} \notin \mathsf{honestTransfers}$.
  (2) $\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}^*, \mathsf{ledger}) = \mathsf{accept}$.
  (3) There exists $\mathsf{tx}_{\mathsf{skeleton}} \in \mathsf{honestTransfers}$ such that:
    (a) $\mathsf{tx}^*_{\mathsf{skeleton}} \neq \mathsf{tx}_{\mathsf{skeleton}}$.
    (b) $\mathsf{tx}^*_{\mathsf{skeleton}}$ and $\mathsf{tx}_{\mathsf{skeleton}}$ share a common nullifier nul.

A remark on change. At first sight, it seems that the definition above does not cover malleability of $\mathsf{note}_{\mathsf{change}}$. In principle, there are two ways that an adversary might tamper with $\mathsf{note}_{\mathsf{change}}$:

- By modifying the recipient of the change. However, note that (npk, R) in $\mathsf{note}_{\mathsf{change}}$ is already specified in $\mathsf{tx}_{\mathsf{skeleton}}$, which the definition does cover.
- By modifying the value of the change. This will not happen, since the value depends deterministically on fee (which is part of $\mathsf{tx}_{\mathsf{skeleton}}$), and $\mathsf{gasSpent}$, which is a public value. Therefore, the adversary cannot modify this value and still have a transaction that is accepted by $\mathsf{VerifyTransaction}$.

Therefore, we conclude that it is enough for our definition to cover $\mathsf{tx}_{\mathsf{skeleton}}$.

2.3. **Ledger indistinguishability.**

Definition 2. *We say that* Π *is* IND*-secure if for any PPT adversary* $\mathcal{A}$*, we have that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND}}(\mathcal{A}) < \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND}}(\mathcal{A})$ *is the advantage of* $\mathcal{A}$ *in the game described below.*

The indistinguishability game IND*.*

- *Setup phase.* The challenger $\mathcal{C}$ runs $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$, sends pp to $\mathcal{A}$. $\mathcal{C}$ initializes two DAP oracles $\mathcal{O}_0, \mathcal{O}_1$, with corresponding ledgers $\mathsf{ledger}_0, \mathsf{ledger}_1$, and samples $b \leftarrow \{0, 1\}$.
- *Query phase.* $\mathcal{A}$ can elicit behavior in the ledgers by submitting pairs of queries (Q, Q'). $\mathcal{C}$ checks that the queries satisfy *public consistency* (defined below). If they do, they are forwarded to the oracles according to the following rules:
  - If the queries are of type **Insert**, $\mathcal{C}$ forwards Q to $\mathcal{O}_b$ and Q' to $\mathcal{O}_{1-b}$.
  - In any other case, $\mathcal{C}$ forwards Q to $\mathcal{O}_0$ and Q' to $\mathcal{O}_1$.

After each query, $\mathcal{C}$ takes the responses a_i from $\mathcal{O}_i$, for $i = 1, 2$, and sends (a_b, a_{1-b}) and (L_b, L_{1-b}) to $\mathcal{A}$.

- *Answer phase.* $\mathcal{A}$ outputs a bit $\tilde{b}$. $\mathcal{A}$ wins iff $b = \tilde{b}$.

Public consistency of queries. A pair of queries (Q, Q') submitted by the adversary in the ledger indistinguishability game must satisfy some public consistency conditions. First, both queries must be of the same type. Furthermore, each type of query has specific requirements.

- **CreateKeys**: both queries must have the same input, and output the same key (same internal randomness).
- **Mint**: both queries must have the same value v.
- **PrepareTransfer**: the queries must be individually well-formed, i.e.
  (1) The input notes referenced by $\mathsf{pos}_i^{\mathsf{old}}$ must be in $\mathsf{noteList}$, and they must not be spent.
  (2) The input addresses $\mathsf{pk}_i^{\mathsf{old}}$ must correspond to the owners of the notes.
  (3) The balance equation is satisfied.

  Furthermore, both queries must be consistent with each other w.r.t. public information and the adversary's view:
  (4) $\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}}$ must be the same in both queries.
  (5) If a recipient address $\mathsf{pk}_i^{\mathsf{new}}$ is not in $\mathsf{independentKeys}$ (i.e. it belongs to the adversary, or the adversary is a helper and therefore knows the corresponding vk), it must be so in both queries. Moreover, $(v, \mathsf{pk})_i^{\mathsf{new}}$ must be the same in both queries.
  (6) If an input note referenced by $\mathsf{pos}_i^{\mathsf{old}}$ was added through an **Insert** query, it must be so in both queries, and the corresponding value v_i^{old} must be the same in both queries.
- **FullTransfer**: same conditions as queries of type **PrepareTransfer**, and additionally:
  (7) $\mathsf{gasSpent}$ must be the same in both queries.
  (8) No transaction resulting from a **FullTransfer** query can contain a nullifier that appeared previously as a result of a **PrepareTransfer** query.[1] Note that such transactions can still be added to the ledgers via **Insert** queries.
- **Receive**: no condition.
- **Insert**: no condition.

2.4. Balance.

Definition 3. *We say that Π is* BAL*-secure if for any PPT adversary $\mathcal{A}$, we have that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}}(\mathcal{A}) < \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}}(\mathcal{A})$ is the advantage of $\mathcal{A}$ in the game described below.

The balance game BAL.

- *Setup phase.* The challenger $\mathcal{C}$ runs $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$, sends pp to $\mathcal{A}$. $\mathcal{C}$ initializes a DAP oracle $\mathcal{O}$, with its corresponding ledger.
- *Query phase.* $\mathcal{A}$ submits queries Q to elicit behavior on honest users. $\mathcal{C}$ forwards Q to $\mathcal{O}$, and relays the answer back to $\mathcal{A}$. There is no special restriction on these queries.

[1]The challenger $\mathcal{C}$ can prevent this from happening by keeping a list of nullifiers produced by **PrepareTransfer** queries, and running **FullTransfer** queries "in the head" before forwarding them to the oracles.

- *Answer phase.* $\mathcal{A}$ signals the end of the experiment. $\mathcal{C}$ computes the following values:
  - v_{spent}: the total amount of payments sent by $\mathcal{A}$ to honest users. $\mathcal{C}$ computes it as follows. Initialize $v_{\mathsf{spent}} = 0$. Then, for each $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk}) \in \mathsf{honestKeys}$:

    (1) Run Scan$(\mathsf{pp}, (\mathsf{vk}, \mathsf{pk}), \mathsf{ledger})$, obtaining $\mathsf{notesFound}$.

    (2) For each $(\mathsf{note}, \mathsf{noteContent}) \in \mathsf{notesFound}$,

      * Let $\mathsf{tx} = (\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent})$ be the transaction that added note to the ledger.[2]

      * Let v be the value of $\mathsf{noteContent}$.

    If $\mathsf{tx}_{\mathsf{skeleton}} \notin \mathsf{honestTransfers}$, add v to v_{spent}.
  - v_{minted}: the total value of notes that $\mathcal{A}$ minted for themselves. $\mathcal{C}$ computes it as follows. Initialize $v_{\mathsf{minted}} = 0$. Then, for each $\mathsf{tx} = (\mathsf{note}, v)$ of type mint in ledger:

    (1) If tx is the result of a **Mint** query, move on to the next iteration.

    (2) For each $\mathsf{pk} \in \mathsf{honestKeys}$, check whether note is owned by pk. To do so, $\mathcal{C}$ runs the following in their head:

      (a) Run **Mint** to get note' of value 0 owned by pk.

      (b) Run **FullTransfer** with input notes $(\mathsf{note}, \mathsf{note}')$, input public key pk, and any valid set of remaining arguments.

    If this results in a valid transfer for any key in $\mathsf{honestKeys}$, move on to the next iteration.

    (3) Add v to v_{minted}.
  - v_{received}: the total amount of payments received by $\mathcal{A}$ from honest users. $\mathcal{C}$ computes it as follows. Initialize $v_{\mathsf{received}} = 0$. Then, for each $\mathsf{tx} = (\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent})$ in ledger such that $\mathsf{tx}_{\mathsf{skeleton}}$ originated from a **PrepareTransfer** or **FullTransfer** query Q:

    (1) For each $\mathsf{note} \in \{\mathsf{note}_1^{\mathsf{new}}, \mathsf{note}_2^{\mathsf{new}}, \mathsf{note}_{\mathsf{change}}\}$:

      (a) Let pk be the argument in Q that corresponds to the public key that owns note.

      (b) Let v be the argument in Q that corresponds to the value of note.[3]

      (c) If $\mathsf{pk} \notin \mathsf{honestKeys}$, add v to v_{received}.

  $\mathcal{A}$ wins iff $v_{\mathsf{spent}} > v_{\mathsf{minted}} + v_{\mathsf{received}}$.

2.5. **Note spendability.**

Definition 4. *We say that* Π *is* NS*-secure if for any PPT adversary* $\mathcal{A}$*, we have that*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{NS}}(\mathcal{A}) < \mathsf{negl}(\lambda),$$

where $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{NS}}(\mathcal{A})$ *is the advantage of* $\mathcal{A}$ *in the game described below.*

The note spendability game NS.

[2] Technically, the same note could appear in different transactions in ledger, e.g. if the sender sends two notes for the same amount to the same recipient, and reuses the randomness. We can make a distinction by looking at the positioned note in the note tree.

[3] In the case of $\mathsf{note}_{\mathsf{change}}$, this value does not appear directly in Q, but can be computed from $\mathsf{gasPrice}, \mathsf{gasLimit}$ and $\mathsf{gasSpent}$.

- *Setup phase.* The challenger $\mathcal{C}$ runs $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$, sends pp to $\mathcal{A}$. $\mathcal{C}$ initializes a DAP oracle $\mathcal{O}$, with its corresponding ledger.
- *First query phase.* $\mathcal{A}$ is allowed to modify the ledger and influence honest users by making queries Q that $\mathcal{C}$ submits to $\mathcal{O}$, sending the answer back to $\mathcal{A}$. There is no special restriction on these queries.
- *Target selection phase.* $\mathcal{A}$ indicates that the first query phase is over, and submits a query

  $$Q^* = (\textbf{PrepareTransfer}, \{(\mathsf{pos}, \mathsf{pk})_i^{\mathsf{old}}, (v, \mathsf{pk})_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}}).$$

  $\mathcal{A}$ receives the answer $(\mathsf{tx}^*_{\mathsf{skeleton}}, \mathsf{publicInputs}^*, \mathsf{secretInputs}^*)$, and this phase ends. If Q^* does not result in a valid transaction, the game is stopped and $\mathcal{A}$ loses.
- *Second query phase.* Same as the first query phase.
- *Answer phase.* $\mathcal{A}$ indicates that the second query phase is over, and sends $\mathsf{gasSpent}$. Let $\mathsf{tx}^* = (\mathsf{tx}^*_{\mathsf{skeleton}}, \mathsf{proof}^*, \mathsf{note}^*_{\mathsf{change}}, \mathsf{gasSpent})$, where
  - $\mathsf{proof}^* = \mathsf{PS.Prove}_{\mathsf{crs}}(\mathsf{publicInputs}^*, \mathsf{secretInputs}^*)$.
  - $\mathsf{note}^*_{\mathsf{change}} = (v^*_{\mathsf{change}}, \triangle, \mathsf{npk}^*_{\mathsf{change}}, R^*_{\mathsf{change}})$, where $(\mathsf{npk}^*_{\mathsf{change}}, R^*_{\mathsf{change}})$ are those contained in fee^*, within $\mathsf{publicInputs}^*$, and $v^*_{\mathsf{change}} = (\mathsf{gasLimit} - \mathsf{gasSpent}) \cdot \mathsf{gasPrice}$.

  $\mathcal{A}$ wins iff $\mathsf{PS.Prove}$ fails or the following three conditions hold:

  (1) $\mathsf{gasSpent} \leq \mathsf{gasLimit}$.

  (2) $\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}^*, \mathsf{ledger}) = \mathsf{reject}$.

  (3) No $\mathsf{pos} \in Q^*$ has been spent in ledger as a result of a transaction that originated from a **PrepareTransfer** or **FullTransfer** query (including Q^*).

3. The Phoenix transaction model

3.1. Data structures.

- ledger containing transactions.
- Merkle tree MT, with notes as leaves.
- $\mathsf{noteContent} = (v, s, \mathsf{pk}, \mathsf{com}, R)$.
- $\mathsf{note} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)$.
- $\mathsf{fee} = (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{npk}, R)$.

3.2. **Algorithms.** Figure 2 contains a description of the algorithm that compose the Phoenix DAP. Refer to the Phoenix review for details on the protocol flow and the transfer circuit description (i.e. the statement being proven with $\mathsf{PS.Prove}$).[4]

The figure contains both the actual protocol (white background), and the modifications made to it in the ledger indistinguishability game (colored background). This is to avoid duplicates in the document. Just ignore the colored bits for the protocol.

[4]The notation might differ slightly from the Phoenix documentation.

```
$\mathsf{Setup}(\lambda)$:
    (1) Sample $\mathbb{J}$ as a cyclic group of order $t$ with $\lambda$ bits of security.
    (2) $G, G' \leftarrow \mathbb{J}$.
    (3) $\mathsf{crs} \leftarrow \mathsf{PS.KeyGen}(\lambda)$.
        $(\mathsf{crs}, \mathsf{trapdoor}) \leftarrow \mathsf{PS.KeyGenExtended}(\lambda)$.
    (4) Output $\mathsf{pp} = (G, G', \mathsf{crs})$.

$\mathsf{CreateKeys}(\mathsf{pp})$:
    (1) $\mathsf{sk} = (a, b) \leftarrow \mathbb{F}_t^2$.
    (2) $\mathsf{pk} = (A, B) = (aG, bG)$.
    (3) $\mathsf{vk} = (a, B)$.
    (4) Output $(\mathsf{sk}, \mathsf{pk}, \mathsf{vk})$.

$\mathsf{Mint}(\mathsf{pp}, v, \mathsf{pk})$:
    (1) Parse $\mathsf{pk} = (A, B)$.
    (2) $r \leftarrow \mathbb{F}_t$.
    (3) $R = rG$.
    (4) $\mathsf{npk} = \mathsf{Hash}^{\mathsf{keys}}(rA)G + B$.
        $\mathsf{npk} \leftarrow \mathbb{J}$.
    (5) $\mathsf{com} = \mathsf{C.Com}(v; 0)$.
    (6) $\mathsf{noteContent} = (v, 0, \mathsf{pk}, \mathsf{com}, R)$.
    (7) $\mathsf{note} = (\mathsf{com}, \triangle, \mathsf{npk}, R)$.
    (8) Output $(\mathsf{noteContent}, \mathsf{note})$.

$\mathsf{VerifyTransaction}(\mathsf{pp}, \mathsf{tx}, \mathsf{ledger})$:
If $\mathsf{tx}$ is of type $\mathsf{mint}$:
    (1) Parse $\mathsf{tx} = ((\mathsf{com}, \triangle, \mathsf{npk}, R), v)$.
    (2) If $\mathsf{C.Com}(v; 0) \neq \mathsf{com}$, output $\mathsf{reject}$, otherwise output $\mathsf{accept}$.
If $\mathsf{tx}$ is of type $\mathsf{transfer}$:
    (1) Parse $\mathsf{tx} = (\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent})$.
    (2) Parse $\mathsf{tx}_{\mathsf{skeleton}} = (\mathsf{root}, \{\mathsf{nul}_i^{\mathsf{old}}\}_{i=1,2}, \{\mathsf{note}_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{fee})$.
    (3) Parse $\mathsf{fee} = (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{npk}_{\mathsf{change}}, R_{\mathsf{change}})$.
    (4) For $i = 1, 2$, parse $\mathsf{note}_i^{\mathsf{new}} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)_i^{\mathsf{new}}$.
    (5) $\mathsf{publicInputs} = \left(G, G', \mathsf{root}, \left\{\mathsf{nul}_i^{\mathsf{old}}\right\}_{i=1,2}, \left\{\mathsf{com}_i^{\mathsf{new}}\right\}_{i=1,2}\right)$.
    (6) Output $\mathsf{reject}$ if any of the following happens:
        • $\mathsf{nul}_1 = \mathsf{nul}_2$ or either of them appears in $\mathsf{ledger}$.
        • $\mathsf{root}$ does not appear on $\mathsf{ledger}$.
        • $\mathsf{note}_{\mathsf{change}} \neq (v_{\mathsf{change}}, \triangle, \mathsf{npk}_{\mathsf{change}}, R_{\mathsf{change}})$, where $v_{\mathsf{change}} = (\mathsf{gasLimit} - \mathsf{gasSpent}) \cdot \mathsf{gasPrice}$.
        • $\mathsf{PS.Verify}_{\mathsf{crs}}(\mathsf{publicInputs}, \mathsf{proof}) \neq \mathsf{accept}$.
        Otherwise, output $\mathsf{accept}$.

$\mathsf{Scan}(\mathsf{pp}, (\mathsf{vk}, \mathsf{pk}), \mathsf{ledger})$:
    (1) Parse $\mathsf{vk} = (a, B)$.
    (2) Parse $\mathsf{pk} = (A, B)$.
    (3) For each $\mathsf{tx}$ in $\mathsf{ledger}$ of type $\mathsf{transfer}$:
        (a) Initialize $\mathsf{notesFound} = \{\}$.
        (b) Parse $\mathsf{tx} = ((\mathsf{root}, \{\mathsf{nul}_i^{\mathsf{old}}, \mathsf{note}_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{fee}), \mathsf{proof})$.
        (c) For $i = 1, 2$:
            (i) Parse $\mathsf{note}_i^{\mathsf{new}} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)_i^{\mathsf{new}}$.
            (ii) $\mathsf{k_{DH}}_i = aR_i^{\mathsf{new}}$.
            (iii) $(v_i^{\mathsf{new}} \mid\mid s_i^{\mathsf{new}}) = \mathsf{E.Dec}_{\mathsf{k_{DH}}_i}(\mathsf{enc}_i^{\mathsf{new}})$. If $\mathsf{E.Dec}$ returns $\perp$, output $\perp$.
            (iv) If $\mathsf{C.Com}(v_i^{\mathsf{new}}; s_i^{\mathsf{new}}) = \mathsf{com}_i^{\mathsf{new}}$:
                (A) Set $\mathsf{noteContent}_i = (v, s, \mathsf{pk}, \mathsf{com}, R)_i^{\mathsf{new}}$.
                (B) Append $(\mathsf{note}, \mathsf{noteContent})$ to $\mathsf{notesFound}$.
    (4) Output $\mathsf{notesFound}$.

$\mathsf{PrepareTransfer}\left(\begin{array}{l}\mathsf{pp}, \mathsf{root}, \mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{pk}_{\mathsf{change}}, \\ \left\{\begin{array}{l}(\mathsf{noteContent}, \mathsf{sk}, \mathsf{pos}, \mathsf{path})_i^{\mathsf{old}} \\ (v, \mathsf{pk})_i^{\mathsf{new}}\end{array}\right\}_{i=1,2}\end{array}\right)$:
    (1) For $i = 1, 2$:
        (a) Parse $\mathsf{noteContent}_i^{\mathsf{old}} = (v, s, \mathsf{pk}, \mathsf{com}, R)_i^{\mathsf{old}}$.
        (b) Parse $\mathsf{sk}_i^{\mathsf{old}} = (a, b)_i^{\mathsf{old}}$.
        (c) $\mathsf{nsk}_i^{\mathsf{old}} = \mathsf{Hash}^{\mathsf{keys}}(a_i^{\mathsf{old}} R_i^{\mathsf{old}}) + b_i^{\mathsf{old}}$.
        (d) $\mathsf{npk}'_i = \mathsf{nsk}_i^{\mathsf{old}} \cdot G'$.
        (e) $\mathsf{nul}_i^{\mathsf{old}} = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}'_i \mid\mid \mathsf{pos})$.
            $\mathsf{nul}_i^{\mathsf{old}} \leftarrow \mathbb{J}$.
        (f) Parse $\mathsf{pk}_i^{\mathsf{new}} = (A, B)_i^{\mathsf{new}}$.
        (g) $r_i^{\mathsf{new}}, s_i^{\mathsf{new}} \leftarrow \mathbb{F}_t$.
        (h) $R_i^{\mathsf{new}} = r_i^{\mathsf{new}} G$.
        (i) $\mathsf{com}_i^{\mathsf{new}} = \mathsf{C.Com}(v_i^{\mathsf{new}}; s_i^{\mathsf{new}})$.
            If $\mathsf{pk}_i^{\mathsf{new}} \in \mathsf{independentKeys}$:
                (i) $\nu \leftarrow \mathsf{C.MessageSpace}$.
                (ii) $\mathsf{com}_i^{\mathsf{new}} = \mathsf{C.Com}(\nu; s_i^{\mathsf{new}})$.
        (j) $\mathsf{noteContent}_i^{\mathsf{new}} = (v, s, \mathsf{pk}, \mathsf{com}, R)_i^{\mathsf{new}}$.
        (k) $\mathsf{k_{DH}}_i = r_i^{\mathsf{new}} A_i^{\mathsf{new}}$.
        (l) $\mathsf{enc}_i = \mathsf{E.Enc}_{\mathsf{k_{DH}}_i}(v_i^{\mathsf{new}} \mid\mid s_i^{\mathsf{new}})$.
            If $\mathsf{pk}_i^{\mathsf{new}} \in \mathsf{independentKeys}$:
                (i) $\mathsf{k_{DH}}_i \leftarrow \mathbb{J}, \mu \leftarrow \mathsf{E.MessageSpace}$.
                (ii) $\mathsf{enc}_i = \mathsf{E.Enc}_{\mathsf{k_{DH}}_i}(\mu)$.
        (m) $\mathsf{npk}_i^{\mathsf{new}} = \mathsf{Hash}^{\mathsf{keys}}(\mathsf{k_{DH}}_i)G + B_i^{\mathsf{new}}$.
            $\mathsf{npk}_i^{\mathsf{new}} \leftarrow \mathbb{J}$.
        (n) $\mathsf{note}_i^{\mathsf{new}} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)_i^{\mathsf{new}}$.
    (2) Parse $\mathsf{pk}_{\mathsf{change}} = (A, B)$.
    (3) $r_{\mathsf{change}} \leftarrow \mathbb{F}_t$.
    (4) $R_{\mathsf{change}} = r_{\mathsf{change}} G$.
    (5) $\mathsf{npk}_{\mathsf{change}} = \mathsf{Hash}^{\mathsf{keys}}(r_{\mathsf{change}} A)G + B$.
        $\mathsf{npk}_{\mathsf{change}} \leftarrow \mathbb{J}$.
    (6) $\mathsf{fee} = (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{npk}_{\mathsf{change}}, R_{\mathsf{change}})$.
    (7) $\mathsf{tx}_{\mathsf{skeleton}} = (\mathsf{root}, \{\mathsf{nul}_i^{\mathsf{old}}, \mathsf{note}_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{fee})$.
    (8) $\mathsf{tx}_{\mathsf{hash}} = \mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}_{\mathsf{skeleton}})$.
    (9) For $i = 1, 2$:
        (a) $\mathsf{sig}_i = \mathsf{S.Sign}_{\mathsf{nsk}_i^{\mathsf{old}}}(\mathsf{tx}_{\mathsf{hash}})$.
    (10) $\mathsf{publicInputs} = \left(G, G', \mathsf{root}, \left\{\mathsf{nul}_i^{\mathsf{old}}, \mathsf{com}_i^{\mathsf{new}}\right\}_{i=1,2}, \mathsf{fee}\right)$.
    (11) $\mathsf{secretInputs} = \left(\left\{\begin{array}{l}(\mathsf{pos}, \mathsf{path}, \mathsf{npk}, \mathsf{npk}', v, s)_i^{\mathsf{old}} \\ (v, s)_i^{\mathsf{new}} \\ \mathsf{sig}_i\end{array}\right\}_{i=1,2}\right)$.
    (12) Output $\left(\begin{array}{l}\{\mathsf{noteContent}_i\}_{i=1,2}, \mathsf{tx}_{\mathsf{skeleton}}, \\ \mathsf{publicInputs}, \mathsf{secretInputs}\end{array}\right)$.

$\mathsf{ProveTransfer}\,(\mathsf{pp}, \mathsf{publicInputs}, \mathsf{secretInputs})$:
    (1) Parse
        $\mathsf{publicInputs} = \left(G, G', \mathsf{root}, \left\{\mathsf{nul}_i^{\mathsf{old}}, \mathsf{com}_i^{\mathsf{new}}\right\}_{i=1,2}, \mathsf{fee}\right)$.
    (2) Parse
        $\mathsf{secretInputs} = \left(\left\{\begin{array}{l}(\mathsf{pos}, \mathsf{path}, \mathsf{npk}, \mathsf{npk}', v, s)_i^{\mathsf{old}} \\ (v, s)_i^{\mathsf{new}} \\ \mathsf{sig}_i\end{array}\right\}_{i=1,2}\right)$.
    (3) $\mathsf{note}_i^{\mathsf{new}} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)_i^{\mathsf{new}}$.
    (4) $\mathsf{tx}_{\mathsf{skeleton}} = (\mathsf{root}, \{\mathsf{nul}_i^{\mathsf{old}}, \mathsf{note}_i^{\mathsf{new}}\}_{i=1,2}, \mathsf{fee})$.
    (5) $\mathsf{proof} = \mathsf{PS.Prove}_{\mathsf{crs}}(\mathsf{publicInputs}, \mathsf{secretInputs})$.
        $\mathsf{proof} = \mathsf{PS.Simulate}_{\mathsf{crs}}(\mathsf{publicInputs}, \mathsf{trapdoor})$.
    (6) Output $\mathsf{proof}$.
```

FIGURE 2. Phoenix algorithms, and their modifications for the ledger indistinguishability sequence of games. Note that all the colored changes in $\mathsf{PrepareTransfer}$ only take place when it is called from within a **FullTransfer** query, but is kept untouched if called from a **PrepareTransfer** query.

4. Security proofs

4.1. **Non-malleability.** Before proving the property, we first observe that, given a Double Schnorr signature with respect to a public key, we can produce a signature of the same message with respect to a different key, as long as we know the relation between the corresponding secret keys (even if we don't know any of them individually).

We recall the Double Schnorr signature scheme:

- $\mathsf{S.Gen}$: sample a secret key $x \leftarrow \mathbb{F}_t$, and set the public key $(X, X') = (xG, xG')$.
- $\mathsf{S.Sign}$: given a message m and a secret key x, sample $\omega \leftarrow \mathbb{F}_t$, set $(\Omega, \Omega') = (\omega G, \omega G')$, compute $c = \mathsf{Hash}(m, \Omega, \Omega')$ and set $u = \omega - cx$. Output the signature $\mathsf{sig} = (\Omega, \Omega', u)$.
- $\mathsf{S.Verify}$: given a message m, a signature $\mathsf{sig} = (\Omega, \Omega', u)$ and a public key (X, X'), compute $c = \mathsf{Hash}(m, \Omega, \Omega')$ and accept iff these two equations hold:
$$uG = \Omega - cX,$$
$$uG' = \Omega - cX'.$$

We now show in detail how to swap keys. Let $x^{\mathsf{old}}, x^{\mathsf{new}} \in \mathbb{F}_t$ be two secret keys, with corresponding public keys $(X, X')^{\mathsf{old}}, (X, X')^{\mathsf{new}}$. Let $\alpha = x^{\mathsf{new}} - x^{\mathsf{old}}$. Then, given any message m and a signature $\mathsf{sig}^{\mathsf{old}} = (\Omega, \Omega', u^{\mathsf{old}})$ of m with respect to $\mathsf{pk}^{\mathsf{old}}$, we can compute
$$\mathsf{sig}^{\mathsf{new}} = \mathsf{S.KeySwap}(\mathsf{sig}^{\mathsf{old}}, \alpha) := (\Omega, \Omega', u^{\mathsf{new}} = u^{\mathsf{old}} - c\alpha).$$
It is easy to see that $\mathsf{sig}^{\mathsf{new}}$ is a valid signature of of m with respect to $(X, X')^{\mathsf{new}}$.

Indeed, observe that $(X, X')^{\mathsf{new}} = (X + \alpha G, X' + \alpha G')^{\mathsf{old}}$. Then
$$u^{\mathsf{new}}G = u^{\mathsf{old}}G - c\alpha G = \Omega - cX^{\mathsf{old}} - c\alpha G = \Omega - c(X^{\mathsf{old}} + \alpha G) = \Omega - cX^{\mathsf{new}},$$
and the same argument works for the second verification equation.

Theorem 1. *The DAP scheme described in Figure 2 is* NM*-secure. More precisely, for any PPT adversary* $\mathcal{A}$*, there exist PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$ *such that:*
$$\mathsf{Adv}^{\mathsf{NM}}_{\mathcal{A}} < \mathsf{Adv}^{\mathsf{SE}}_{\mathcal{B}_1} + \mathsf{Adv}^{\mathsf{CR-nul}}_{\mathcal{B}_2} + \mathsf{Adv}^{\mathsf{CR-sig}}_{\mathcal{B}_3} + q_{\mathbf{CreateKeys}} \cdot \mathsf{Adv}^{\mathsf{EU-CMA}}_{\mathcal{B}_4},$$
where

- $\mathsf{Adv}^{\mathsf{SE}}_{\mathcal{B}_1}$ *is the advantage of* $\mathcal{B}_1$ *in breaking the simulation extractability of* $(\mathsf{PS.KeyGen}, \mathsf{PS.Prove}, \mathsf{PS.Verify})$.
- $\mathsf{Adv}^{\mathsf{CR-nul}}_{\mathcal{B}_2}$ *is the advantage of* $\mathcal{B}_2$ *in breaking the collision resistance of* $\mathsf{Hash}^{\mathsf{nul}}$.
- $\mathsf{Adv}^{\mathsf{CR-sig}}_{\mathcal{B}_3}$ *is the advantage of* $\mathcal{B}_3$ *in breaking the collision resistance of* $\mathsf{Hash}^{\mathsf{sig}}$.
- $\mathsf{Adv}^{\mathsf{EU-CMA}}_{\mathcal{B}_4}$ *is the advantage of* $\mathcal{B}_4$ *in breaking the* $\mathsf{EU-CMA}$ *property of* $(\mathsf{S.Gen}, \mathsf{S.Sign}, \mathsf{S.Verify})$.

Proof. Suppose that $\mathcal{A}$ wins the NM game. We start by observing that, since $\mathsf{tx}_{\mathsf{skeleton}} \in \mathsf{honestTransfers}$, the challenger $\mathcal{C}$ knows the corresponding $\mathsf{secretInputs}$. This contains, in particular, the positions pos_i for $i = 1, 2$ of the notes being spent. Let $J \subset \{1, 2\}$ be the set of indices such that $j \in J$ iff $\mathsf{nul}^*_j = \mathsf{nul}_j$. The set J is non-empty by condition 3b of Definition 1.

On the other hand, $\mathcal{C}$ does not directly know the $\mathsf{secretInputs}^*$ corresponding to tx^*, but they can try to extract it from proof^* by means of $\mathsf{PS.Extract}$. If $\mathsf{PS.Extract}$ succeeds, $\mathcal{C}$ learns $\mathsf{secretInputs}^*$, which contains $(\mathsf{pos}^*, \mathsf{npk}^*, \mathsf{npk}'^*)^{\mathsf{old}}_i$ for $i = 1, 2$. Consider the following events:

- E_1: $\mathcal{A}$ wins the NM game and $\mathsf{PS.Extract}$ produces $\mathsf{secretInputs}^*$ that is not a valid witness for $\mathsf{publicInputs}^*$.

- E_2: $\mathcal{A}$ wins the NM game, E_1 does not happen and $(\mathsf{npk}_j{'}^*, \mathsf{pos}_j^*) \neq (\mathsf{npk}_j', \mathsf{pos}_j)$ for at least one $j \in J$.
- E_3: $\mathcal{A}$ wins the NM game, E_1, E_2 do not happen and $\mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}^*_{\mathsf{skeleton}}) = \mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}_{\mathsf{skeleton}})$.
- E_4: $\mathcal{A}$ wins the NM game, E_1, E_2, E_3 do not happen and $\mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}^*_{\mathsf{skeleton}}) \neq \mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}_{\mathsf{skeleton}})$.

Clearly,

$$\Pr[\mathcal{A} \text{ wins the } \mathsf{NM} \text{ game}] = \Pr[E_1] + \Pr[E_2] + \Pr[E_3] + \Pr[E_4].$$

We study each of these cases separately.

E_1. In this case, it is easy to build a reduction $\mathcal{B}_1$ that breaks the simulation extractability property of the proof system. $\mathcal{B}_1$ takes the role of challenger in the NM game against $\mathcal{A}$.[5] If E_1 happens, then $\mathsf{PS.Extract}(\mathsf{proof}^*)$ allows $\mathcal{B}_1$ to find a pair $(\mathsf{publicInputs}^*, \mathsf{secretInputs}^*)$ that is not a valid assignment for the transfer circuit. Therefore $\Pr[E_1] \leq \mathsf{Adv}^{\mathsf{SE}}_{\mathcal{B}_1}$.

E_2. From this point onward, we can use that $\mathsf{PS.Extract}(\mathsf{proof}^*)$ produces a valid witness $\mathsf{secretInputs}^*$. We build a reduction $\mathcal{B}_2$ that uses $\mathcal{A}$ to break the collision resistance of $\mathsf{Hash}^{\mathsf{nul}}$. $\mathcal{B}_2$ acts as the challenger in the NM game against $\mathcal{A}$. If E_2 happens, let $j \in J$ such that $(\mathsf{npk}_j{'}^*, \mathsf{pos}_j^*) \neq (\mathsf{npk}_j', \mathsf{pos}_j)$. We also have that $\mathsf{nul}_j^* = \mathsf{nul}_j$. Recall that $\mathsf{nul}_j = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}_j', \mathsf{pos}_j)$. Thus, we have that

$$\mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}_j{'}^*, \mathsf{pos}_j^*) = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}_j', \mathsf{pos}_j),$$

where $(\mathsf{npk}_j{'}^*, \mathsf{pos}_j^*) \neq (\mathsf{npk}_j', \mathsf{pos}_j)$. $\mathcal{B}_2$ has found a collision in $\mathsf{Hash}^{\mathsf{nul}}$, and therefore $\Pr[E_2] \leq \mathsf{Adv}^{\mathsf{CR-nul}}_{\mathcal{B}_2}$.

E_3. We build a reduction $\mathcal{B}_3$ that uses $\mathcal{A}$ to break the collision resistance of $\mathsf{Hash}^{\mathsf{sig}}$. This case is very similar to E_2 above. In E_3, we have that $\mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}^*_{\mathsf{skeleton}}) = \mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}_{\mathsf{skeleton}})$, but by condition 3a of Definition 1, $\mathsf{tx}^*_{\mathsf{skeleton}} \neq \mathsf{tx}_{\mathsf{skeleton}}$. Thus, $\mathcal{B}_3$ has found a collision in $\mathsf{Hash}^{\mathsf{sig}}$, hence $\Pr[E_3] \leq \mathsf{Adv}^{\mathsf{CR-sig}}_{\mathcal{B}_3}$.

E_4. We use $\mathcal{A}$ to build a reduction $\mathcal{B}_4$ that breaks the existential unforgeability of the signature scheme. $\mathcal{B}_4$ receives from the $\mathsf{EU-CMA}$ challenger $\mathcal{C}$ a signature public key (X, X'). During a randomly chosen **CreateKeys** query, $\mathcal{B}_4$ chooses $a \leftarrow \mathbb{F}_t$ and sets $A = aG$ as usual, but sets $B = X, B' = X'$. $\mathcal{B}_4$ sends $\mathsf{pk} = (A, B)$ or $\mathsf{vk} = (a, B)$ to $\mathcal{A}$, as requested.

Whenever $\mathcal{A}$ makes a query of type **PrepareTransfer** or **FullTransfer** with input pk, $\mathcal{B}_4$ can compute everything on their own (note that a, B, B' are enough to compute $\mathsf{npk}, \mathsf{npk}'$), except for the signature $\mathsf{sig} = \mathsf{S.Sign}_{\mathsf{nsk}}(\mathsf{tx}_{\mathsf{hash}})$, which would normally require the corresponding nsk, unknown to $\mathcal{B}_4$. Instead, $\mathcal{B}_4$ asks $\mathcal{C}$ for a signature $\tilde{\mathsf{sig}} = \mathsf{S.Sign}_{\mathsf{sk}}(\mathsf{tx}_{\mathsf{hash}})$. Now, since $\mathcal{B}_4$ knows the difference $\alpha = \mathsf{nsk} - b = H(rA)$, they can run $\mathsf{S.KeySwap}(\tilde{\mathsf{sig}}, \alpha)$ to obtain a valid signature of $\mathsf{tx}_{\mathsf{hash}}$ signed with nsk. All other queries are answered as usual.

After the experiment ends, let $\mathsf{tx}_{\mathsf{skeleton}} \in \mathsf{honestTransfers}$ such that $\mathsf{tx}^*_{\mathsf{skeleton}}$ and $\mathsf{tx}_{\mathsf{skeleton}}$ share their ith nullifiers. $\mathcal{B}_4$ checks whether the ith note spent by $\mathsf{tx}_{\mathsf{skeleton}}$ was owned by pk. If that is not the case, $\mathcal{B}_4$ aborts.

Otherwise, because E_1, E_2, E_3 do not happen, $\mathsf{secretInputs}^*$ contains $(\mathsf{npk}_i^*, \mathsf{npk}_i{'}^*, \mathsf{sig}_i^*)$ for all $i = 1, 2$, such that:

(1) $\mathsf{S.Verify}_{(\mathsf{npk}_i^*, \mathsf{npk}_i{'}^*)}(\mathsf{Hash}(\mathsf{tx}^*_{\mathsf{skeleton}}), \mathsf{sig}_i^*) = \mathsf{accept}$ for all $i = 1, 2$.

(2) $\mathsf{npk}_j{'}^* = \mathsf{npk}_j'$ and $\mathsf{pos}_j^* = \mathsf{pos}_j$ for all $j \in J \neq \emptyset$.

(3) $\mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}^*_{\mathsf{skeleton}}) \neq \mathsf{Hash}^{\mathsf{sig}}(\mathsf{tx}_{\mathsf{skeleton}})$.

[5]Note that knowledge soundness is not enough, since $\mathcal{A}$ expects to see other proofs when making **FullTransfer** queries, so we require simulation extractability.

At this point, $\mathcal{B}_4$ has a forgery of a message for which they have not queried $\mathcal{C}$, but with respect to the wrong key. More precisely, sig_j^* is a valid signature for $\mathsf{tx}^*_{\mathsf{skeleton}} \notin \mathsf{honestTransfers}$ with respect to the key $(\mathsf{npk}_j^*, \mathsf{npk}_j'^*) = (\mathsf{npk}_j, \mathsf{npk}_j')$, which has appeared in the query that produced $\mathsf{tx}_{\mathsf{skeleton}}$. Thus, $\mathcal{B}_4$ knows the corresponding $\alpha^* = b_j - \mathsf{nsk}_j = -H(r_j A_j)$, and therefore send $\mathsf{S.KeySwap}(\mathsf{sig}_j^*, \alpha^*)$ to $\mathcal{C}$, winning the $\mathsf{EU-CMA}$ game. Accounting for the probability of aborting, we conclude that $\Pr[E_4] \leq q_{\mathbf{CreateKeys}} \cdot \mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{EU-CMA}}$. □

4.2. **Ledger indistinguishability.** Given an adversary $\mathcal{A}$ against IND, let:

- $q_{\mathbf{CreateKeys}}$ be the number of queries of type **CreateKeys**.
- $q_{\mathbf{Mint}}$ be the number of queries of type **Mint**.
- $q_{\mathbf{FullTransfer}}$ be the number of queries of type **FullTransfer**.

Theorem 2. *The DAP scheme described in Figure 2 is* IND*-secure. More precisely, for any PPT adversary* $\mathcal{A}$*, there exist PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ *such that:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND}} < 2 \cdot q_{\mathbf{FullTransfer}} \cdot (\mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{IND-CCA}} + \mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{DDH}} + \mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Hiding}}) + \mathsf{negl}(\lambda),$$

where

- $\mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{IND-CCA}}$ *is the advantage of* $\mathcal{B}_1$ *in breaking the* $\mathsf{IND-CCA}$ *property of* $(\mathsf{E.Gen}, \mathsf{E.Enc}, \mathsf{E.Dec})$.
- $\mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{DDH}}$ *is the advantage of* $\mathcal{B}_2$ *in breaking the* DDH *problem.*
- $\mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Hiding}}$ *is the advantage of* $\mathcal{B}_3$ *in breaking the* Hiding *property of* $(\mathsf{C.Gen}, \mathsf{C.Com})$.

We prove the theorem through a sequence of indistinguishable security games. The modifications described in PrepareTransfer only take place when it is called from within a **FullTransfer** query, but does not change if called from a **PrepareTransfer** query.

- Game_0: the real security game, as described in Figure 2, ignoring all the text highlighted in colors.
- Game_1 : same as Game_0, with modifications marked with this . We simulate the transfer proof.
- Game_2 : same as Game_1, with modifications marked with this . We replace the ciphertexts addressed to honest users that do not have the adversary as a helper by ciphertext of a random message under a random key.
- Game_3 : same as Game_2, with modifications marked with this . We replace the output of hashes by random elements.
- Game_4 : same as Game_3, with modifications marked with this . We replace commitments to note values by commitments to random values.

Given an adversary $\mathcal{A}$ and a game Game, we denote by $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}}$ the advantage of $\mathcal{A}$ in winning Game.

Lemma 3. *For any PPT adversary* $\mathcal{A}$, $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_1} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_0}\right| = 0.$

Proof. The only difference between the games is that we simulate the transfer proofs. Thus, this result follows from the perfect zero-knowledge property of the proof system. □

Lemma 4. *For any PPT adversary* $\mathcal{A}$*, there exist PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2$ *such that*

$$\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_2} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_1}\right| \leq 2 \cdot q_{\mathbf{FullTransfer}} \cdot (\mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{IND-CCA}} + \mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{DDH}}).$$

Proof. We define an intermediate game $\mathbf{H}$, in which we replace the plaintext by a random message but keep the keys intact. More precisely, in FullTransfer queries, for $i = 1, 2$, if $\mathsf{pk}_i^{\mathsf{new}} \in \mathsf{independentKeys}$, $\mathbf{H}$ replaces $\mathsf{enc}_i = \mathsf{E.Enc}_{\mathsf{k_{DH}}_i}(v_i^{\mathsf{new}} \mid\mid s_i^{\mathsf{new}})$ by $\mathsf{enc}_i = \mathsf{E.Enc}_{\mathsf{k_{DH}}_i}(\mu_i)$, where μ_i is a uniformly random element of the message space.

We also define a sequence of hybrid games from Game_1 to $\mathbf{H}$. For $j = 0, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$, let $\mathsf{Game}_{1,j}$ be the game in which the modification described above involves the j first ciphertexts. Clearly $\mathsf{Game}_{1,0} = \mathsf{Game}_1$, and $\mathsf{Game}_{1,2\cdot q_{\mathbf{FullTransfer}}} = \mathbf{H}$. Our next step is to bound the advantage in distinguishing between two adjacent games $\mathsf{Game}_{1,j-1}$ and $\mathsf{Game}_{1,j}$, for $j = 1, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$.

We show how to use $\mathcal{A}$ to build an attacker $\mathcal{B}$ against the IND-CCA property of the encryption scheme. $\mathcal{B}$ interacts with $\mathcal{A}$, following $\mathsf{Game}_{1,j-1}$, except when $\mathcal{A}$ makes a query of type FullTransfer which would produce the jth ciphertext. Let $\mathsf{pk}^{\mathsf{new}}$ be the public key associated to the jth ciphertext.

- If $\mathsf{pk}^{\mathsf{new}} \notin \mathsf{independentKeys}$, the ciphertext does not change between the two games, and thus they behave in exactly the same way, so the advantage in distinguishing between them is 0.
- If $\mathsf{pk}^{\mathsf{new}} \in \mathsf{independentKeys}$, let $m_0 = (v^{\mathsf{new}} \mid\mid s^{\mathsf{new}})$ be the message that would be encrypted under $\mathsf{Game}_{1,j-1}$, and let $m_1 \leftarrow \mathsf{E.MessageSpace}$. $\mathcal{B}$ forwards (m_0, m_1) to the IND-CCA challenger $\mathcal{C}$, who answers with an encryption enc of m_b, for $b \leftarrow \{0,1\}$. $\mathcal{B}$ embeds enc in the query response as the jth ciphertext, and continues the experiment as usual. When $\mathcal{A}$ finally outputs $\tilde{b}$, $\mathcal{B}$ forwards it to $\mathcal{C}$ as their response. It is straightforward to see that, when $b = 0$, $\mathcal{A}$ is playing $\mathsf{Game}_{1,j-1}$, and when $b = 1$, $\mathcal{A}$ is playing $\mathsf{Game}_{1,j}$.

Thus, $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_{1,j}} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_{1,j-1}}\right| \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND-CCA}}$, and by composing these changes over the $2 \cdot q_{\mathbf{FullTransfer}}$ ciphertexts, we get that $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathbf{H}} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_1}\right| \leq 2 \cdot q_{\mathbf{FullTransfer}} \cdot \mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND-CCA}}$.

What remains now is to go from $\mathbf{H}$ to Game_2. To do so, again, we define a sequence of hybrid games. For $j = 0, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$, let $\mathbf{H}_j$ be as $\mathbf{H}$, except that, for the first j ciphertexts, if $\mathsf{pk}_i^{\mathsf{new}} \in \mathsf{independentKeys}$, we replace $\mathsf{k_{DH}}_i = r_i^{\mathsf{new}} A_i^{\mathsf{new}}$ by a uniform element in $\mathbb{J}$. Note that this also affects the computation of $\mathsf{npk}_i^{\mathsf{new}}$, to keep things consistent.

We have that $\mathbf{H}_0 = \mathbf{H}$ and $\mathbf{H}_{2\cdot q_{\mathbf{FullTransfer}}} = \mathsf{Game}_2$. Again, we will bound the advantage in distinguishing between two adjacent games $\mathbf{H}_{j-1}$ and $\mathbf{H}_j$, for $j = 1, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$.

We now use $\mathcal{A}$ to build an attacker $\mathcal{B}$ against the DDH problem in $\mathbb{J}$. $\mathcal{B}$ interacts with $\mathcal{A}$ following $\mathbf{H}_{j-1}$, with the following modifications. Fixed a generator G, The DDH challenger $\mathcal{C}$ sends to $\mathcal{B}$ a challenge $(X, Y, Z) \in \mathbb{J}^3$, where $X = xG, Y = yG$ and Z might be either xyG or a random element of $\mathbb{J}$. Again, we want to embed the challenge from $\mathcal{C}$ in the jth ciphertext. Let $\mathsf{pk}^{\mathsf{new}}$ be the public key associated to the jth ciphertext.

- If $\mathsf{pk}^{\mathsf{new}} \notin \mathsf{independentKeys}$, the two games are equal and the advantage in distinguishing is 0.
- If $\mathsf{pk}^{\mathsf{new}} \in \mathsf{independentKeys}$, then $\mathcal{A}$ must have made a CreateKeys query that resulted in $\mathsf{pk}^{\mathsf{new}}$. During that query, $\mathcal{B}$ sets $\mathsf{pk}^{\mathsf{new}} = (A, B)$, where $A = X$ and B is computed as usual.[6] Then, when answering the FullTransfer query that involves the jth ciphertext, $\mathcal{B}$ sets the corresponding $R_i^{\mathsf{new}} = Y$ and $\mathsf{k_{DH}}_i = Z$. The simulation continues as usual, and the final response $\tilde{b}$ from $\mathcal{A}$ is forwarded to $\mathcal{C}$. Now, when $b = 0$, $\mathcal{A}$ is playing $\mathbf{H}_{j-1}$, and when $b = 1$, $\mathcal{A}$ is playing $\mathbf{H}_j$.

[6]At first, this seems backwards, as $\mathcal{B}$ must answer the CreateKeys query before knowing which public key $\mathcal{A}$ will use in the jth ciphertext. Actually, what $\mathcal{B}$ can do is to embed the key provided by $\mathcal{C}$ in different CreateKeys queries, re-running $\mathcal{A}$ with the same randomness and a different CreateKeys query in case of a mismatch. This works because, at most, $\mathcal{B}$ has to run $\mathcal{A}$ $q_{\mathbf{CreateKeys}}$ times.

Thus, $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathbf{H}_j} - \mathsf{Adv}_{\mathcal{A}}^{\mathbf{H}_{j-1}}\right| \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}}$. Iterating over all the ciphertexts, we deduce that

$$\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_2} - \mathsf{Adv}_{\mathcal{A}}^{\mathbf{H}}\right| \leq 2 \cdot q_{\mathbf{FullTransfer}} \cdot \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DDH}},$$

which concludes the proof. □

Lemma 5. *For any PPT adversary $\mathcal{A}$, $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_3} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_2}\right| \leq \mathsf{negl}(\lambda)$ in the random oracle model.*[7]

Proof. In the random oracle model, all the outputs of hashes already behave as random elements in the corresponding codomain. We just need to quantify the probability that two inputs across all queries are the same.

In Game_2, we have:

(1) $\mathsf{npk} = \mathsf{Hash}(rA)G + B$ for $r \leftarrow \mathbb{F}_t$, both in **Mint** and **FullTransfer** queries.

(2) $\mathsf{nul}_i^{\mathsf{old}} = \mathsf{Hash}(\mathsf{npk}_i' \mid\mid \mathsf{pos})$ for $i = 1, 2$ in **FullTransfer** queries.

The second type always has unique input, due to pos being unique. Therefore, we just need to bound the probability that two values of r are the same among all $q^* = q_{\mathbf{Mint}} + 3 \cdot q_{\mathbf{FullTransfer}}$ instances of npk (in **FullTransfer** queries, we count both output notes and the change note). This is a case of the birthday problem, and thus the probability of collision is bounded by

$$1 - \prod_{j=1}^{q^*} \left(\frac{t-j}{t}\right) \leq 1 - \left(1 - \frac{q^*}{t}\right)^{q^*} = \mathsf{poly}\left(\frac{q^*}{t}\right),$$

where $\deg(\mathsf{poly}) = q^*$. Since q^* is polynomial in λ and t is exponential in λ, we conclude that $\mathsf{poly}(q^*/t) \leq \mathsf{negl}(\lambda)$. □

Lemma 6. *For any PPT adversary $\mathcal{A}$, there exists a PPT algorithm $\mathcal{B}_3$ such that*

$$\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_4} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_3}\right| \leq 2 \cdot q_{\mathbf{FullTransfer}} \cdot \mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Hiding}}.$$

Proof. The proof is very similar to the proof of Lemma 4 above. We define a sequence of hybrid games. For $j = 0, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$, let $\mathsf{Game}_{1,j}$ be the game in which we replace the first j commitments in **FullTransfer** queries. Clearly $\mathsf{Game}_{3,0} = \mathsf{Game}_3$, and $\mathsf{Game}_{3,2\cdot q_{\mathbf{FullTransfer}}} = \mathsf{Game}_4$. For $j = 1, \ldots, 2 \cdot q_{\mathbf{FullTransfer}}$, we bound the advantage in distinguishing between $\mathsf{Game}_{3,j-1}$ and $\mathsf{Game}_{3,j}$.

We show how to use $\mathcal{A}$ to build an attacker $\mathcal{B}$ against the hiding property of the commitment scheme. $\mathcal{B}$ receives the commitment key from the Hiding challenger. $\mathcal{B}$ interacts with $\mathcal{A}$, following $\mathsf{Game}_{3,j-1}$, except when $\mathcal{A}$ makes a query of type **FullTransfer** which would produce the jth commitment. At this point, this query contains a value v_i^{new}. Set $m_0 = v_i^{\mathsf{new}}$ and $m_1 \leftarrow \mathsf{C.MessageSpace}$. $\mathcal{B}$ forwards (m_0, m_1) to the Hiding challenger, who answers with a commitment com of m_b, where $b \in \{0, 1\}$. $\mathcal{B}$ embeds com in the query response, as the jth commitment, and continues the experiment as usual. When $\mathcal{A}$ finally outputs $\tilde{b}$, $\mathcal{B}$ forwards it to $\mathcal{C}$ as their response. It is straightforward to see that, when $b = 0$, $\mathcal{A}$ is playing $\mathsf{Game}_{1,j-1}$, and when $b = 1$, $\mathcal{A}$ is playing $\mathsf{Game}_{1,j}$. □

[7]Proving this step in the random oracle model should not be necessary. However, proving it in the standard model requires rephrasing the hashes as PRFs and some careful analysis (and possibly some small modifications to the protocol), so for now at least we have a proof in the ROM. A proof in the standard model would probably depend on the unpredictability of the outputs of $\mathsf{Hash}^{\mathsf{keys}}$ and $\mathsf{Hash}^{\mathsf{nul}}$.

Lemma 7. *For any adversary $\mathcal{A}$, $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_4} = 0$.*

Proof. We argue that the responses of query pairs (Q, Q') from the oracles and their modifications to the ledgers do not depend on the secret bit, and thus provide no help at all to the adversary $\mathcal{A}$. Hence the conclusion that $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}_4} = 0$. To argue this, we examine each type of query.

- **CreateKeys**: as described in the experiment, it provides the same response to both Q, Q'.
- **Mint**: each response produces a npk that is not linked with anything, and both responses contain a commitment to the same value.
- **PrepareTransfer**: since the nullifier of a note is deterministic, public consistency condition 8 implies that the adversary cannot call **FullTransfer** on any of the same input notes. Thus, transactions produced through **PrepareTransfer**, or other transactions with the same input notes, can only be written to the ledgers via **Insert** queries, which do not help $\mathcal{A}$ (see below).
- **FullTransfer**: each query writes in the corresponding ledger a transaction containing

$$\begin{pmatrix} \mathsf{root}, \{\mathsf{nul}_i^{\mathsf{old}}, (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)_i^{\mathsf{new}}\}_{i=1,2}, (\mathsf{gasPrice}, \mathsf{gasLimit}, \mathsf{npk}_{\mathsf{change}}, R_{\mathsf{change}}), \\ \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent} \end{pmatrix}.$$

  The proof is simulated and does not contain any secret information. $\mathsf{nul}_i^{\mathsf{old}}, \mathsf{npk}_{\mathsf{change}}$ are random group elements. The remaining information depends only on the query inputs $(v, \mathsf{pk})_i^{\mathsf{new}}$.

  - If $\mathsf{pk}_i^{\mathsf{new}} \notin \mathsf{independentKeys}$, public consistency condition 5 implies that these inputs are the same, and thus the adversary has no advantage in distinguishing.
  - If $\mathsf{pk}_i^{\mathsf{new}} \in \mathsf{independentKeys}$, we have that $\mathsf{npk}_i^{\mathsf{new}}$ and R_i^{new} are random group elements; $\mathsf{com}_i^{\mathsf{new}}$ is a commitment to a random value; and $\mathsf{enc}_i^{\mathsf{new}}$ is a ciphertext of a random value under a random key. Nothing here provides any information whatsoever for $\mathcal{A}$.
- **Receive**: produces no new information for the adversary.
- **Insert**: the transactions are inserted in order $(b, 1-b)$, so this does not help $\mathcal{A}$ in guessing b.

□

4.3. **Balance.** We define $\mathsf{ledgerAugmented}$ as a list of tuples $(\mathsf{tx}, \mathsf{secretInputs})$, where $\mathsf{tx} \in \mathsf{ledger}$, and

$$\mathsf{secretInputs} = \left(\left\{ \begin{array}{l} (\mathsf{pos}, \mathsf{path}, \mathsf{npk}, \mathsf{npk}', v, s)_i^{\mathsf{old}} \\ (v, s)_i^{\mathsf{new}} \\ \mathsf{sig}_i \end{array} \right\}_{i=1,2} \right)$$

is computed by $\mathcal{C}$ as follows:

- In the case of queries of type **Mint**, **PrepareTransfer** and **FullTransfer**, $\mathcal{C}$ simply saves the secret data from the oracle call.
- In the case of queries of type **Insert**:
  - If tx is of type mint, $\mathcal{C}$ saves the public information only.
  - If tx is of type $\mathsf{transfer}$, $\mathcal{C}$ runs $\mathsf{PS.Extract}$ on the proof contained in tx.

Definition 5. *Let* $\mathsf{ledgerAugmented} = \{(\mathsf{tx}, \mathsf{secretInputs})\}_{\mathsf{tx} \in \mathsf{ledger}}$, *where in particular*

$$(\mathsf{npk}, \mathsf{pos}, v, s)_i^{\mathsf{old}} \subset \mathsf{secretInputs},$$

for $i = 1, 2$. We say that $\mathsf{ledgerAugmented}$ *is* balanced *if the following conditions are satisfied:*

(1) Notes spent exist in the ledger: for all transfers $(\mathsf{tx}, \mathsf{secretInputs}) \in \mathsf{ledgerAugmented}$ *and all* $i = 1, 2$, $(v; s)_i^{\mathsf{new}}$ *is an opening of the commitment at* $\mathsf{pos}_i^{\mathsf{old}}$ *in* $\mathsf{MT}_{\mathsf{ledger}}$ *has appeared previously in* ledger*, as part of an output note.*

(2) Notes spent are unique: no two transfers $(\mathsf{tx}_0, \mathsf{secretInputs}_0), (\mathsf{tx}_1, \mathsf{secretInputs}_1) \in \mathsf{ledgerAugmented}$ *contain a common* pos*, nor the same transaction contains the same position twice.*

(3) Balance is preserved within each transfer: for all $(\mathsf{tx}, \mathsf{secretInputs}) \in \mathsf{ledgerAugmented}$ *of type* $\mathsf{transfer}$*, we have that*

$$\sum_{i=1,2} v_i^{\mathsf{old}} = \sum_{i=1,2} v_i^{\mathsf{new}} + \mathsf{gasPrice} \cdot \mathsf{gasSpent} + v_{\mathsf{change}}.$$

(4) Notes are consistent across transactions: for all transfers $(\mathsf{tx}, \mathsf{secretInputs}) \in \mathsf{ledgerAugmented}$ *and all* $i = 1, 2$*, let* tx' *be the transaction in which the commitment at* $\mathsf{pos}_i^{\mathsf{old}}$ *in* $\mathsf{MT}_{\mathsf{ledger}}$ *was added.*

- *If* $\mathsf{tx}' = (\mathsf{note}, v)$ *is of type* mint*, then* $v_i^{\mathsf{old}} = v$.
- *If* tx' *is of type* $\mathsf{transfer}$*, then* tx' *contains an output* $\mathsf{note}' = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)$ *that was added to* MT *at position* $\mathsf{pos}_i^{\mathsf{old}}$*, and* com *had an opening* (v, s)*. We have that:*

  (a) $v_i^{\mathsf{old}} = v$.

  (b) If tx *has been added to* ledger *via* **Insert***, then* note' *is not received when running* $\mathsf{Scan}(\mathsf{pp}, (\mathsf{vk}, \mathsf{pk}), \mathsf{ledger})$ *for any* $(\mathsf{vk}, \mathsf{pk})$ *in* $\mathsf{honestKeys}$.

Theorem 8. *The DAP scheme described in Figure 2 is* BAL*-secure. More precisely, for any PPT adversary* $\mathcal{A}$*, there exist PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4, \mathcal{B}_5$ *such that:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}} \le q_{\mathbf{Insert}} \cdot \mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{SE}} + \mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{Soundness-MT}} + 2 \cdot \mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Soundness-sig}} + \mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{Binding}} + \mathsf{Adv}_{\mathcal{B}_5}^{\mathsf{NM}},$$

where

- $\mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{SE}}$ *is the advantage of* $\mathcal{B}_1$ *in breaking the simulation extractability of* PS.
- $\mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{Soundness-MT}}$ *is the advantage of* $\mathcal{B}_2$ *in breaking the soundness of the* MT *vector commitment scheme.*
- $\mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Soundness-sig}}$ *is the advantage of* $\mathcal{B}_3$ *in breaking the soundness of the signature scheme (as a proof of same discrete logarithm).*
- $\mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{Binding}}$ *is the advantage of* $\mathcal{B}_4$ *in breaking the binding property of the commitment scheme.*
- $\mathsf{Adv}_{\mathcal{B}_5}^{\mathsf{NM}}$ *is the advantage of* $\mathcal{B}_5$ *in breaking the non-malleability of the DAP scheme.*

Proof. We start by switching to a security game BAL' that is the same as BAL, except that $\mathcal{C}$ maintains $\mathsf{ledgerAugmented}$ instead of ledger, and $\mathcal{A}$ can also win if $\mathsf{PS.Extract}$ fails in any of its invocations. It is clear that the adversary cannot distinguish between the two games unless $\mathsf{PS.Extract}$ fails, so there exists $\mathcal{B}_1$ such that $\left|\mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}'} - \mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}}\right| \le q_{\mathbf{Insert}} \cdot \mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{SE}}$.

We will next show that an adversary $\mathcal{A}$ winning the BAL' game necessarily means that they managed to make the ledger unbalanced. More precisely, for $i = 1, \ldots, 4$, let E_i be the following event:

(1) $\mathcal{A}$ wins the BAL' game.

(2) E_j does not happen for any $j < i$.

(3) $\mathsf{ledgerAugmented}$ does not satisfy the ith property of Definition 5.

We will first argue that

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{BAL}} \leq \Pr[E_1] + \Pr[E_2] + \Pr[E_3] + \Pr[E_4],$$

and then complete the proof by bounding each of these terms separately. To prove the former, we show that if $\mathcal{A}$ plays the BAL' game while following all conditions, then

$$v_{\mathsf{spent}} \leq v_{\mathsf{minted}} + v_{\mathsf{received}}. \tag{1}$$

We do so by induction on the number of queries. Clearly, at the beginning of the experiment, both sides of the inequality are 0. Assume that, after $n-1$ queries, the inequality still holds. We argue that any nth query that respects the conditions of Definition 5 does not break inequality (1). The outcome is different depending on the type of query:

- **CreateKeys**, **PrepareTransfer** or **Receive**: none of the values in inequality (1) change.
- **Mint**: the corresponding pk must be in $\mathsf{honestKeys}$, and thus none of the values in inequality (1) change.
- **FullTransfer**: this only allows $\mathcal{A}$ to make honest users spend their notes. Thus:
  - If all $\mathsf{pk}_i^{\mathsf{new}}$ are in $\mathsf{honestKeys}$, then nothing changes.
  - If some $\mathsf{pk}_i^{\mathsf{new}}$ is not in $\mathsf{honestKeys}$, then v_{received} increases by the corresponding v_i^{new}, but the inequality still holds.
- **Insert**: we are in one of these two situations:
  - $\mathsf{tx} = (\mathsf{note}, v)$ is of type mint. Then:
    - If note can be received via Scan by any pk in $\mathsf{honestKeys}$, then nothing changes.
    - Otherwise, v_{minted} increases by v.
  - tx is of type $\mathsf{transfer}$: this is the only way in which $\mathcal{A}$ can increase v_{spent}, by producing notes that can be received by some $\mathsf{pk} \in \mathsf{honestKeys}$. The conditions of Definition 5 imply the following:
    - (1) implies that notes must appear in ledger to be available to spend.
    - (2) implies that each positioned note can only be spent once.
    - (4b) implies that $\mathcal{A}$ can only spend their own notes.

    These three conditions imply that the maximal amount of value that $\mathcal{A}$ has available is $v_{\mathsf{minted}} + v_{\mathsf{received}}$. On the other hand:
    - (4a) imply that the value of a note does not change from the moment it was created to the moment it was spent.
    - (3) implies that the value contained in the notes created in a transaction cannot exceed the value of the notes spent.

    Together, these conditions imply that $\mathcal{A}$ cannot make honest users receive more money than what $\mathcal{A}$ has available to spend. Putting it all together, we conclude that

$$v_{\mathsf{spent}} \leq v_{\mathsf{minted}} + v_{\mathsf{received}}.$$

Now that we have ensured that E_1, E_2, E_3, E_4 cover all possible scenarios, all that remains is to bound the probability of each of them happening, which we do in the following lemmas. □

Lemma 9. *For any PPT adversary $\mathcal{A}$, there exists $\mathcal{B}_2$ such that* $\Pr[E_1] \le \mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{Soundness-MT}}$.

Proof. We build a reduction $\mathcal{B}_2$ that uses $\mathcal{A}$ to break the soundness of the MT vector commitment scheme. $\mathcal{B}_2$ simulates the environment for $\mathcal{A}$ until the BAL' game is finished. At this point, because E_1 happens, there exists $(\mathsf{tx}, \mathsf{secretInputs}) \in \mathsf{ledgerAugmented}$ such that, for some $i = 1, 2$, $(v; s)_i^{\mathsf{new}}$ is not an opening of the commitment at position $\mathsf{pos}_i^{\mathsf{old}}$. However, because $\mathsf{secretInputs}$ is a valid witness for tx, it holds that $\mathsf{MT.Verify}(\mathsf{C.Com}(v; s), \mathsf{pos}, \mathsf{path}) = \mathsf{accept}$. Thus, $\mathcal{B}_2$ has found an accepting proof for an element that is not in the tree. □

Lemma 10. *For any PPT adversary $\mathcal{A}$, there exists $\mathcal{B}_3$ such that* $\Pr[E_2] \le 2 \cdot \mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{Soundness-sig}}$.

Proof. We build a reduction $\mathcal{B}_3$ that uses $\mathcal{A}$ to break the soundness property of the signature scheme. That is, $\mathcal{B}_3$ will attempt to produce an accepting signature with respect to a pair $(\mathsf{npk}, \mathsf{npk}')$ such that

$$\mathsf{Dlog}_G(\mathsf{npk}) \ne \mathsf{Dlog}_{G'}(\mathsf{npk}'). \tag{2}$$

Note that, in the BAL' game, no nullifier can appear twice, or else the transaction will be rejected. Thus, if E_2 happens, it must be because two nullifiers $\mathsf{nul}_0, \mathsf{nul}_1$ are published, such that

$$\mathsf{nul}_0 = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}'_0, \mathsf{pos}), \qquad \mathsf{nul}_1 = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}'_1, \mathsf{pos}),$$

for a common position pos and $\mathsf{npk}'_0 \ne \mathsf{npk}'_1$. Furthermore, $\mathsf{secretInputs}_0, \mathsf{secretInputs}_1$ contain signatures $\mathsf{sig}_0, \mathsf{sig}_1$, respectively, such that

$$\mathsf{S.Verify}_{(\mathsf{npk}_0, \mathsf{npk}'_0)}(\mathsf{tx}_{\mathsf{skeleton},0}, \mathsf{sig}_0) = \mathsf{accept},$$
$$\mathsf{S.Verify}_{(\mathsf{npk}_1, \mathsf{npk}'_1)}(\mathsf{tx}_{\mathsf{skeleton},1}, \mathsf{sig}_1) = \mathsf{accept},$$

for some $(\mathsf{npk}_0, \mathsf{npk}'_0) \in \mathsf{secretInputs}_0$ and some $(\mathsf{npk}_1, \mathsf{npk}'_1) \in \mathsf{secretInputs}_1$. Because E_1 does not happen, pos refers to a position in MT that contains $\mathsf{note} = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)$, and because the transfer proofs corresponding to nul_0 and nul_1 are accepting, necessarily $\mathsf{npk}_0 = \mathsf{npk} = \mathsf{npk}_1$. Thus, at this point, we have obtained two pairs $(\mathsf{npk}, \mathsf{npk}'_1)$ and $(\mathsf{npk}, \mathsf{npk}'_1)$, with $\mathsf{npk}'_0 \ne \mathsf{npk}'_1$, that pass verification. However, for a fixed npk, there is only one possible npk' such that the same discrete logarithm relation of equation (2) holds. Thus, once of the two signatures must be a forgery. $\mathcal{B}_3$ picks $b \leftarrow 0, 1$ and sends sig_b to the challenger. With probability 1/2, they have chosen the forgery. □

Lemma 11. *For any PPT adversary $\mathcal{A}$,* $\Pr[E_3] = 0$.

Proof. In this case, because $\mathcal{A}$ is playing the BAL' game, the extraction of a valid witness for each transfer has already succeeded, so condition (3) always holds. □

Lemma 12. *For any PPT adversary $\mathcal{A}$, there exist $\mathcal{B}_4, \mathcal{B}_5$ such that* $\Pr[E_4] \le \mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{Binding}} + \mathsf{Adv}_{\mathcal{B}_5}^{\mathsf{NM}}$.

Proof. We split E_4 into two sub-events:

- $E_{4,1}$: $\mathcal{A}$ breaks the condition on mint transactions or condition (4a).
- $E_{4,2}$: $\mathcal{A}$ breaks condition (4b).

We bound the probability of these events separately.

$E_{4,1}$. We build a reduction $\mathcal{B}_4$ that breaks the binding property of the commitment scheme. $\mathcal{B}_4$ simulates the environment for $\mathcal{A}$.

Assume that tx spends a note that was added to ledger in a mint transaction $\mathsf{tx}' = (\mathsf{note}, v)$, at position $\mathsf{pos}_i^{\mathsf{old}}$, but $v_i^{\mathsf{old}} \ne v$. Because tx' was accepted into the ledger, we have that $\mathsf{com}_i^{\mathsf{old}} = \mathsf{C.Com}(v; 0)$. On the other hand, because the transfer proof of tx is accepting and the extractor has succeeded, we have that $\mathsf{com}_i^{\mathsf{old}} = \mathsf{C.Com}(v_i^{\mathsf{old}}; s_i^{\mathsf{old}})$. Thus, $\mathcal{B}_4$ has two different openings of the same commitment, breaking the binding property.

Consider now the case in which tx spends a note that was added to ledger in a $\mathsf{transfer}$ transaction tx' that added note' at $\mathsf{pos}_i^{\mathsf{old}}$, but $v_i^{\mathsf{old}} \neq v$. Because the transfer proof of tx' is accepting, we know that $\mathsf{com} = \mathsf{C.Com}(v; s)$. On the other hand, because the transfer proof of tx is accepting, we know that $\mathsf{com} = \mathsf{C.Com}(v_i^{\mathsf{old}}; s_i^{\mathsf{old}})$. Again, $\mathcal{B}_4$ has found two openings for the same commitment.

$E_{4,2}$. In this situation, there is a transfer tx added to ledger via **Insert** that spends a $\mathsf{note}' = (\mathsf{com}, \mathsf{enc}, \mathsf{npk}, R)$ such that $\mathsf{npk} = H(aR)G + B$, where (a, B) is a view key in $\mathsf{honestKeys}$. We use this to build a reduction $\mathcal{B}_5$ to the non-malleability property.

$\mathcal{B}_5$ simulates the environment for $\mathcal{A}$ in the BAL' game, by making use of the oracles provided to them in the NM game. When $E_{4,2}$ happens and tx is inserted in ledger, $\mathcal{B}_5$ stops the BAL' experiment. Then they query the NM challenger exactly in the same way that $\mathcal{A}$ has queried $\mathcal{B}_5$, except that they stop just before inserting tx. Instead, they make a **PrepareTransfer** query that spends in an honest way the same notes that tx would spend. $\mathcal{B}_5$ knows these because they control $\mathsf{ledgerAugmented}$ in the BAL' game that $\mathcal{A}$ is playing. Finally, they submit tx, which shares a common nullifier with the transaction produced via the **PrepareTransfer** query. Thus, $\mathcal{B}_5$ wins the NM game. □

4.4. Note spendability.

Theorem 13. *The DAP scheme described in Figure 2 is* NS*-secure. More precisely, for any PPT adversary* $\mathcal{A}$*, there exist PPT algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$ *such that:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{NS}} < \mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{Completeness}} + q_{\mathbf{Insert}} \cdot \mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{SE}} + \mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{CR-nul}} + q_{\mathbf{CreateKeys}} \cdot \mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{EU-CMA}},$$

where

- $\mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{Completeness}}$ *is the advantage of* $\mathcal{B}_1$ *in breaking the completeness of* PS.
- $\mathsf{Adv}_{\mathcal{B}_2}^{\mathsf{SE}}$ *is the advantage of* $\mathcal{B}_2$ *in breaking the simulation extractability of* PS.
- $\mathsf{Adv}_{\mathcal{B}_3}^{\mathsf{CR-nul}}$ *is the advantage of* $\mathcal{B}_3$ *in breaking the collision resistance of* $\mathsf{Hash}^{\mathsf{nul}}$.
- $\mathsf{Adv}_{\mathcal{B}_4}^{\mathsf{EU-CMA}}$ *is the advantage of* $\mathcal{B}_4$ *in breaking the* $\mathsf{EU-CMA}$ *property of* $(\mathsf{S.Gen}, \mathsf{S.Sign}, \mathsf{S.Verify})$.

Proof. Assume that $\mathcal{A}$ wins the NS game. We split the probability of $\mathcal{A}$ winning them game in different cases. Consider the following events:

- E_1: $\mathcal{A}$ wins and $\mathsf{PS.Prove}$ fails in the answer phase, or produces proof^* such that
$$\mathsf{PS.Verify}_{\mathsf{crs}}(\mathsf{publicInputs}^*, \mathsf{proof}^*) \neq \mathsf{accept}.$$
- E_2: $\mathcal{A}$ wins, E_1, E_2 do not happen, and a common nullifier appears in tx^* and ledger.

By condition (2) in Definition 4, tx^* does not pass verification. Given that tx^* was honestly generated, inspection of the VerifyTransaction algorithm shows that rejection can only happen if the proof is rejected, $\mathsf{note}^*_{\mathsf{change}}$ does not match the intended value, or the nullifier is already in the ledger. However, $\mathsf{note}^*_{\mathsf{change}}$ has been honestly computed, and the only thing the adversary has control over is $\mathsf{gasSpent}^*$. Condition (1) from Definition 4 ensures that this is not an issue. Thus, $\Pr[\mathcal{A} \text{ wins}] = \Pr[E_1]\Pr[E_2]$. We now bound each of these probabilities.

E_1. Given that $\mathsf{tx}^*_{\mathsf{skeleton}}$ was honestly computed, the corresponding $(\mathsf{publicInputs}^*, \mathsf{secretInputs}^*)$ pair is a valid pair of statement and witness for the transfer circuit. Hence, an honestly generated proof^* will be accepted unless the completeness property of PS fails. Therefore $\Pr[E_1] \leq \mathsf{Adv}_{\mathcal{B}_1}^{\mathsf{Completeness}}$.

E_2. The transaction tx^* was valid at the time of the target selection phase. Hence, if a nullifier from tx^* already appears in ledger at the end of the experiment, it must have been added during the second query phase. $\mathcal{C}$ keeps a list $\mathsf{pre\text{-}nullifiers}$, composed of of the secret data used in the queries, in particular the pairs $(\mathsf{npk}', \mathsf{pos})$ used to compute nullifiers in all queries of the second query phase.

(1) For queries of types **PrepareTransfer** or **FullTransfer**, $\mathcal{C}$ simply stores this information from the query input and computation.

(2) For queries of type **Insert**, $\mathcal{C}$ runs $\mathsf{PS.Extract}$ on the corresponding proof. When the extractor succeeds, we obtain the pairs $(\mathsf{npk}', \mathsf{pos})$ involved in this case too.

We break E_2 further into different sub-events.

- $E_{2,1}$: E_2 happens and, for some $i = 1, \dots, q_{\mathbf{Insert}}$, the algorithm $\mathsf{PS.Extract}$ does not return a valid witness for the corresponding transfer.
- $E_{2,2}$: E_2 happens, $E_{2,1}$ does not happen, and no pair $(\mathsf{npk}'^*, \mathsf{pos}^*)$ in tx^* (and its corresponding $\mathsf{secretInputs}^*$) is in $\mathsf{pre\text{-}nullifiers}$.
- $E_{2,3}$: E_2 happens, $E_{2,1}, E_{2,2}$ do not happen, and a pair $(\mathsf{npk}'^*, \mathsf{pos}^*)$ in tx^* (and its corresponding $\mathsf{secretInputs}^*$) is in $\mathsf{pre\text{-}nullifiers}$.

Clearly, $\Pr[E_2] = \Pr[E_{2,1}] + \Pr[E_{2,2}] + \Pr[E_{2,3}]$. We bound the probability of each event individually.

$E_{2,1}$. It is straightforward to build a reduction $\mathcal{B}_2$ to the simulation extractability property of PS. $\mathcal{B}_2$ chooses $i = 1, \dots, q_{\mathbf{Insert}}$ uniformly at random, and acts as a challenger for $\mathcal{A}$, simulating its environment and resorting to the proving oracle for proofs. If $\mathsf{PS.Extract}$ fails on the ith **Insert** query, $\mathcal{B}_2$ wins. Thus, accounting for the random choice of i, we have that $\Pr[E_{2,1}] \leq q_{\mathbf{Insert}} \cdot \mathsf{Adv}^{\mathsf{SE}}_{\mathcal{B}_2}$.

$E_{2,2}$. We build a reduction $\mathcal{B}_3$ that uses $\mathcal{A}$ to break the collision resistance of $\mathsf{Hash}^{\mathsf{nul}}$. Let nul be the nullifier that appears in tx^* and in some transaction $\mathsf{tx} = (\mathsf{tx}_{\mathsf{skeleton}}, \mathsf{proof}, \mathsf{note}_{\mathsf{change}}, \mathsf{gasSpent})$ that was added to the ledger in the second query phase. Let $\mathsf{npk}'_1, \mathsf{pos}_1$ be the values used to compute nul in Q^*, known by $\mathcal{B}_3$. Let $\mathsf{npk}'_2, \mathsf{pos}_2$ be the values used to compute nul in $\mathsf{tx}_{\mathsf{skeleton}}$, which $\mathcal{B}_3$ has extracted from proof. Because the extractor succeeded, we have that

$$\mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}'_1, \mathsf{pos}_1) = \mathsf{Hash}^{\mathsf{nul}}(\mathsf{npk}'_2, \mathsf{pos}_2),$$

where $(\mathsf{npk}'_1, \mathsf{pos}_1) \neq (\mathsf{npk}'_2, \mathsf{pos}_2)$. $\mathcal{B}_3$ has found a collision of $\mathsf{Hash}^{\mathsf{nul}}$, and thus $\Pr[E_{2,2}] \leq \mathsf{Adv}^{\mathsf{CR}}_{\mathcal{B}_3}$.

$E_{2,3}$. Let pos be the position in Q^* that also appears in some query Q in the second query phase, and let npk' be the corresponding value that is used together with pos to compute the common nullifier nul. Due to condition (3) of Definition 4, Q must be of type **Insert**, and no queries of type **PrepareTransfer** could have involved pos. Essentially, this means that the adversary has been able to spend an honestly owned note in a dishonest way. We will use this fact to build a reduction $\mathcal{B}_4$ that uses $\mathcal{A}$ to break the $\mathsf{EU-CMA}$ property of the signature scheme. The technique is exactly the same as in the $\mathsf{EU-CMA}$ reduction in the proof of non-malleability (Theorem 1), which we briefly recall here for completeness.

$\mathcal{B}_4$ receives from the $\mathsf{EU-CMA}$ challenger $\mathcal{C}$ a signature public key (X, X'). During a randomly chosen **CreateKeys** query, $\mathcal{B}_4$ sets the public key and view key as $\mathsf{pk} = (A, B), \mathsf{vk} = (a, B)$, with a, A computed as usual, but embedding the challenge X into B.

Whenever $\mathcal{A}$ makes a query of type **PrepareTransfer** or **FullTransfer** with input pk, $\mathcal{B}_4$ queries $\mathcal{C}$ for $\mathsf{sig} = \mathsf{S.Sign}_{\mathsf{sk}}(\mathsf{tx}_{\mathsf{hash}})$ and computes the rest by themselves. Since the signature is with respect to sk instead of nsk, $\mathcal{B}_4$ runs $\mathsf{S.KeySwap}$ to obtain a signature with respect to nsk.

After the experiment ends, let $\mathsf{tx} \in \mathsf{ledger}$ such that tx^* and tx have a common nullifier nul. $\mathcal{B}_4$ checks whether the note spent by tx^* that corresponds to nul was owned by pk. If that is not the case, $\mathcal{B}_4$ aborts.

Otherwise, at this point $\mathcal{B}_4$ has a forgery of a message for which they have not queried $\mathcal{C}$ (this is implied by the conditions of $E_{2,3}$, as discussed above), but with respect to the wrong key. Again, $\mathcal{B}_4$ uses $\mathsf{S.KeySwap}$ to undo the change and recover a forgery with respect to sk. In conclusion, we have that $\Pr[E_{2,3}] \leq q_{\mathbf{CreateKeys}} \cdot \mathsf{Adv}^{\mathsf{EU-CMA}}_{\mathcal{B}_4}$. □

5. Differences with Zerocash

5.1. **Protocol.**

(1) Some differences in terminology with respect to Zerocash [BSCG+14], summarized in the following table.

Zerocash	Phoenix
Coin	Note
Pour	Transfer
Receive	Scan
Serial number	Nullifier

(2) The CreateKeys algorithm outputs a third key, the view key, which is not present in Zerocash. In Phoenix, it is used in Scan, as explained below.

(3) The Transfer (Pour) algorithm in Zerocash has been split into two parts:

- PrepareTransfer: computes the transaction data and signs, using the sender's sk.
- ProveTransfer: takes the transaction data and produces a proof. Uses some secret data but not the sender's sk. This models the action of a third party helper that computes the proof on behalf of the sender.

(4) The Mint and PrepareTransfer algorithms are slightly different from Zerocash, as the form of a note is different. Furthermore, in PrepareTransfer, we first sign and then prove, whereas in Zerocash they first produce proofs to then sign and send the signatures.

(5) Two differences in the Scan (Receive) algorithm:

(a) We use vk instead of sk.

(b) Because of this, this algorithm cannot compute nullifiers (since these depend on npk', for which b is required). Thus, the algorithm does not ensure that a retrieved note can be spent. This motivated the name change.

This should not lead to problems in Phoenix, since unlike in Zerocash, the sender has no control over the pre-nullifier, due to the position being involved. This is captured in the note spendability property, which did not exist in Zerocash.

(6) We make changes throughout the protocol to accommodate for gas and change. More precisely, the PrepareTransfer algorithm now computes an additional stealth address in which to receive the change. When a transaction goes through, a new note is minted with respect to the stealth address chosen by the sender. This is done in the open, so the only secret information is the static public key of the sender.

5.2. **Oracle queries.**

(1) Queries of type **CreateKeys** return vk in addition to pk. This models the fact that the adversary might be acting as a network-listening helper to the honest users.

(2) Queries of type **Mint** now start with the check that $\mathsf{pk} \in \mathsf{honestKeys}$, which does not happen in Zerocash. This means that the **Mint** query allows the adversary to mint notes for honest users, but not for themselves. However, we argue that this is not a restriction, since the adversary can still mint notes through **Insert** queries. Moreover, in Zerocash, **Mint** queries addressed to the adversary themselves result in notes that cannot be spent, because the adversary does not learn ρ. Thus, our restriction does not make the adversary any weaker.

The reason for this change is to have Lemma 5 not depend on $q_{\mathbf{Mint}}$, as all values replaced in **Mint** depend on a key that was generated through **CreateKeys**.[8]

(3) **Transfer** queries have been replaced by two types of queries:

- **FullTransfer**: they take the role of **Transfer** queries in Zerocash, in which the adversary influences the behavior of honest users, and the result is reflected on the ledger.
- **PrepareTransfer**: similarly, these allow the adversary to influence honest users in sending transfers. The key difference is that these essentially output the unproven transaction ($\mathsf{tx}_{\mathsf{skeleton}}$) and the data necessary to prove it. Crucially, these do not modify the ledger. This kills two birds with one stone.
  - These queries allow us to model the scenario in which the adversary acts as a proof helper to the honest sender, who provides all secret data so that the adversary can compute the transfer proof and add the transaction to the ledger (via an **Insert** query) on their behalf.
  - They also provide a suitable interface for the note spendability game, in which the adversary is able to see transactions before they are added to the ledger. This is meant to model roadblock attacks.

  Note that this query creates a proof. This is only so that we can verify the transaction (e.g. notes at $\mathsf{pos}_i^{\mathsf{old}}$ have not been spent before), but this proof is discarded.

(4) **FullTransfer** queries now include check for excessive gas consumption and mint the change note. This allows to model gas in security games. Note that the adversary can choose the gas data, including $\mathsf{gasSpent}$, and $\mathsf{pk}_{\mathsf{change}}$.

5.3. **Non-malleability.** Definition:

(1) The list $\mathsf{honestTransfers}$ maintained by $\mathcal{C}$ contains transfers that come from both **PrepareTransfer** and **FullTransfer** queries, instead of just **Transfer** queries. The intuition is that $\mathcal{A}$ plays against all the honest transactions, and their goal is to modify any of them, even when $\mathcal{A}$ themselves are the proof helper.

(2) $\mathsf{honestTransfers}$ contains only transaction skeletons (transactions signed but not proven). This is because those coming from **PrepareTransfer** actually do not contain a proof, since such proof is later computed by $\mathcal{A}$, who has access to $\mathsf{publicInputs}$ and $\mathsf{secretInputs}$. This is related to the fact that in Zerocash they take a Prove-and-Sign approach, whereas we opt for a Sign-and-Prove approach.

(3) We add the condition $\mathsf{tx}^* \notin \mathsf{honestTransfers}$ for $\mathcal{A}$ to win. This is actually necessary to avoid a trivially broken model, because $\mathcal{A}$ can produce variations of the same transaction (i.e. transactions with the same effect, or with at least a shared nullifier) through **PrepareTransfer** queries. This issue dos not arise in Zerocash because their transactions that originate from any type of query are added instantly to the ledger. Intuitively, our modification does not go against the spirit of the definition, as what we are trying to achieve is to prevent $\mathcal{A}$ from inducing effects on the ledger not intended by honest transactions. And if $\mathsf{tx}^*_{\mathsf{skeleton}} \in \mathsf{honestTransfers}$, it might be different as a transaction but produces the same affects as an honest one, so no harm is done.

(4) We modify the verification condition, replacing the prefix of ledger up to tx by the whole ledger at the end of the experiment. The reason why they went with the prefix in Zerocash was to

[8]Currently, Lemma 5 relies on the ROM. The comment above refers to a future version in the standard model, in which we model the hashes as PRFs.

model the notion that $\mathcal{A}$ might intercept a transaction before it is added to the ledger. However, we already model this capability through **PrepareTransfer**, which essentially produces transfers from honest users (instigated by $\mathcal{A}$), but we don't add them to ledger yet.

Proof:

(1) Due to differences between Phoenix and Zerocash, the pieces of the two proofs are not in a one-to-one correspondence. Regardless, the overall approach is very similar, and roughly rely on the same assumptions.

(2) We require simulation extractability from PS.

5.4. **Ledger indistinguishability.** Definition:

(1) In the public consistency condition, and subsequently through the proof, we replace honestKeys by independentKeys to restrict ourselves only to those users that do not have the adversary as a scan helper.

(2) In the public consistency condition, when one of the output addresses of a **FullTransfer** query is not in independentKeys, we not only require that the value is the same, as in Zerocash, but we also require that the address is the same.

(3) In the public consistency condition, **PrepareTransfer** and **FullTransfer** share mostly the same conditions, inherited from those on **Transfer** in Zerocash. However, **FullTransfer** additionally requires that the nullifiers it produces did not appear before in **PrepareTransfer** queries.[9] Intuitively, this condition is justified by the fact that, if an honest user asks for the adversary's help in producing a proof, they must provide all secret data of the transfer, so clearly the adversary will be able to distinguishing the resulting transaction from another.

(4) We incorporate anonymity of the change recipient into the property. Note that a way for $\mathcal{A}$ to win the game would be to distinguish the npk to which change was addressed, even when choosing different $\mathsf{pk}_{\mathsf{change}}$ in both ledgers.

Proof:

(1) In the sequence of games, rather than always changing the behavior of PrepareTransfer in all queries, we only do so in when it happens inside of a **FullTransfer** query. The reason for this is that, intuitively, the transactions that result from direct **PrepareTransfer** queries are "adversarially controlled",[10] in the sense that the adversary has the secret information, and thus they would be able to distinguish between games.

(2) In Game_4, we do not replace the commitments in **Mint** queries. Zerocash did that because their commitments contained the identity of the recipient, which is what might differ between the two queries of a given pair Q, Q'. However, our commitment in Mint only contains the value, which has to be the same for both queries anyway. This is not the case in PrepareTransfer, hence why we do swap the commitments in **FullTransfer** queries.

5.5. **Balance.** Definition:

(1) Since there is no basecoin or separate public pool in Phoenix, we get rid of v_{Basecoin} in the balance equation. One might think that we need to account for the change with a value replacing this one. However, the change should not be accounted for in the balance equation,

[9]Note that this could not happen in the Zerocash model, since **Transfer** automatically writes new transactions on the ledger, and thus transfers with repeated nullifiers would be rejected. In contrast, transfers coming from **PrepareTransfer** are not added to the ledger immediately.

[10]Moreover, we treat them similarly to transactions created by the adversary in the head, as they can only end up in the ledgers through **Insert** queries.

since it is value that has not been transferred from one party to another. We show how, assuming that v_{change} appears in the equation, the model is trivially broken.

(a) Start from an empty ledger, i.e. all values are 0 in the balance equation.

(b) $\mathcal{A}$ mints a note of value 1 for themselves. v_{mint} becomes 1.

(c) $\mathcal{A}$ creates an honest user.

(d) $\mathcal{A}$ inserts a transaction in which they transfer a value 0 to the honest user. The change is 1, and v_{change} becomes 1.

(e) $\mathcal{A}$ inserts a transaction in which they transfer a value 1 to the honest user. The change is 0, and $v_{\mathcal{A}\to\mathsf{honestKeys}}$ becomes 1.

At this point, the value in the LHS of the balance equation is 2, whereas the RHS value is 1. Thus, balance is broken.

(2) We have compresssed v_{Mint} (minted value for all users) and v_{Unspent} (unspent minted value for honest users) from Zerocash into v_{minted} (minted value for the adversary). The idea is to check whether each minted note is owned by an honest user, and in that case it is not counted.

Proof:

(1) Because of differences in the protocol, the proof is structured in a slightly different way, but overall follows the same approach from Zerocash. Most notably, the conditions for a balanced augmented ledger are different. This is because we do not have uniqueness of commitments, and have to rely on the position in the Merkle tree to identify notes.

5.6. **Note spendability.** Definition:

(1) This property is missing in Zerocash, and was introduced in [GH19]. Another flavor of the definition was later discussed in [Hop22]. Our definition is inspired by the latter, but trying to get it more in tune with the Zerocash style of definitions (followed in the other three properties), and adapted to the specifics of Phoenix. The intuition is that the adversary interacts with a ledger, chooses an honest transaction tx^* as a target that is not yet in the ledger, and attempts to modify the ledger to prevent tx^* from being valid anymore.

Proof:

(1) The proof is similar to the proof of non-malleability, and loosely follows the corresponding proof of note spendability in [GH19].

References

[BSCG⁺14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014. 2, 20

[GH19] Ariel Gabizon and Daira Hopwood. A security analysis of the Zcash Sapling Protocol. 2019. https://github.com/zcash/sapling-security-analysis. 2, 23

[Hop22] Daira Hopwood. Understanding the Security of Zcash. 2022. https://github.com/daira/zcash-security. Slides from a talk given at Zcon3. 2, 23