



Article

On the Security of the Dandelion Protocol

Brian Goncalves ^{*,†}  and Atefeh Mashatan [†] 

Cybersecurity Research Lab, Ryerson University, Toronto, ON M5B 2K3, Canada; amashatan@ryerson.ca

* Correspondence: bgoncalves@ryerson.ca

† These authors contributed equally to this work.

Abstract: In this paper, we review the peer-to-peer blockchain transaction protocol, Dandelion, and develop an oracle-based model for its network and security. We formalize a series of security and functional criteria, such as unforgeability, non-repudiation, and immutability, into security experiments. In our model, we consider a quantum-capable adversary who seeks to undermine any of the security criteria while using oracles to simulate and interact with the Dandelion network. We then prove the security of Dandelion in our model with a series of (tight) security reductions as our main result. In addition, we prove that Dandelion is resistant to double-spending attacks.

Keywords: blockchain; post-quantum; network security



Citation: Goncalves, B.; Mashatan, A. On the Security of the Dandelion Protocol. *Mathematics* **2022**, *10*, 1054. <https://doi.org/10.3390/math10071054>

Academic Editors: Ioana Boureanu and Liqun Chen

Received: 24 February 2022

Accepted: 22 March 2022

Published: 25 March 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Since the release of the Bitcoin paper by Nakamoto [1], blockchain technology has experienced rapid growth in both academic and industry interest. Numerous protocols, such as Ethereum [2], Ripple [3], Phantom [4], HyperLedger [5], and Ouroboros Praos [6], have presented different approaches to the decentralized/distributed nature of blockchain technologies and digital currencies. These protocols present blockchain solutions with different modifications and restrictions on who may join the network, participate, or submit blocks.

While the above represents a small sample of the more famous protocols, the development and adoption is likely to continue to grow in the longer term due to the significant interest from industry. In their 2020 global blockchain survey, Deloitte polled nearly 1400 executives across 14 countries, and 53% responded that blockchain technology was listed among their top strategic priorities [7]. Moreover, nearly a third stated that their company was in development of a blockchain-related project, and 84% indicated that they already use blockchain technology to some extent. Lastly, in their survey, Deloitte estimated that global spending on blockchain solutions will exceed USD 11 billion. Furthermore, the application of blockchain has spread beyond just financial use as a result of extended research conducted on its application to electronic health records [8,9], supply chain management [10,11], and smart cities [12].

These figures and estimates demonstrate that blockchain technology will continue to expand and become increasingly prevalent globally. As such, there is, then, a matching need for these protocols to be provably secure and trustworthy in the long-term. This last criteria of long-term security and trustworthiness is especially vital with the advancements in quantum computers. The existence of Shor's algorithm [13] and the Grover search [14] undermine the security of currently deployed cryptographic algorithms. Shor's algorithm provides quantum computers the ability to recover the secret keys used for both encryption and signing for both RSA and ECC systems. The Grover search can be used to find collisions in hash functions, which are used to ensure the immutability of the blockchain. Thus, any current, and more importantly, future blockchain protocols must consider the potential of quantum-based attacks.

1.1. Our Contributions

In this paper, we present formal proofs of security for the Dandelion blockchain protocol. The Dandelion protocol is designed to be a mobile peer-to-peer transaction application that allows for asynchronous payment requests and deposits [15]. More specifically, we review the protocol and define a set of properties necessary for both functionality and security in the asynchronous, distributed network of Dandelion. To do so, we define a series of black-box oracles that would allow a (potentially quantum) adversary to simulate the entire network, and a set of security experiments that capture our defined properties. Briefly, these criteria include fault tolerance, decidability, unforgeability, non-repudiation, immutability, and double-spending. We then present direct, tight security proofs against a quantum-computing-capable adversary for each of the outlined criteria.

1.2. Paper Organization

The organization of this paper is as follows. In Section 2, we review the necessary background notation and cryptographic algorithms used in this paper and the Dandelion protocol. In Section 3, we outline the network model, a description of users, a summary of Dandelion's message flows, and a brief discussion of the key differences between Dandelion and other protocols. In Section 4, we define our security model for Dandelion, including a formal description of adversarial powers, via oracles and goals. In Section 5, we then prove both the functional and security properties of Dandelion in our model. We conclude our paper in Section 6. In Appendix A, we provide a detailed description of how messages are handled. In Appendix B, we describe how out-of-sync nodes are dealt with in our oracle model.

2. Preliminaries

In this section, we introduce the notation used in this paper, and the necessary cryptographic algorithms used in the Dandelion protocol. We will first review the notation for (cryptographic) algorithms, sampling, and negligible functions.

This section introduces the notation used in this paper and the necessary cryptographic algorithms used in the Dandelion protocol. First, we will review the notation for (cryptographic) algorithms, sampling, and negligible functions.

2.1. Notation

By $y \leftarrow \mathcal{A}(x)$ we denote an algorithm, \mathcal{A} , that runs on input x and output y . When \mathcal{A} has access to an oracle, \mathcal{B} , that it may query, we write this as $\mathcal{A}(x)^{\mathcal{B}(\cdot)}$. If \mathcal{A} is an algorithm that uses some randomness in its execution on input x and we wish to specify what the randomness is, say r , we denote it as $\mathcal{A}(x; r)$. We will refer to specific subroutines within $\mathcal{A}(\cdot)$ as \mathcal{A} .Subroutine. We will consider all adversaries as probabilistic polynomial time (PPT) algorithms on their input length.

We write $x \leftarrow_S S$ to denote that x was outputted by S probabilistically, where if S is some algorithm, then x was selected according to some internal distribution, and if S is some space, such as $\{0, 1\}^l$, then we implicitly mean for x to be sampled uniformly at random.

We say a function g mapping non-negative integers to non-negative reals is called *negligible*, if for all positive numbers c there exists an integer $\lambda_0(c) \geq 0$, such that for all $\lambda > \lambda_0(c)$, we have $g(\lambda) < \frac{1}{\lambda^c}$.

2.2. Digital Signatures

Next, we outline the core asymmetric cryptographic component of Dandelion.

Definition 1 (Digital Signature Algorithms). We say a triple of algorithms $\Sigma = (\text{KeyGen}, \text{Sign}, \text{Verify})$ form a digital signature algorithm (DSA) scheme, if:

- **KeyGen:** The key generation algorithm is a probabilistic algorithm which on input 1^k ($k \in \mathbb{N}$) outputs a related pair, (pk, sk) , of public verification and secret signing keys;

- **Sign:** The signing algorithm is a probabilistic algorithm that takes two inputs, a secret signing key sk , and a plaintext message m , from a designated message space, \mathcal{M}_Σ , and outputs a signature σ ;
- **Verify:** The decryption algorithm is a deterministic algorithm that takes as input a public verification key pk , a plaintext message m , and a signature σ , and returns either a 0 if rejected, or a 1 if accepted.

We now define our security notions for DSAs.

Definition 2 (EUF-CMA Security for DSAs). We say that a DSA, Σ , is EUF-CMA-secure if, for all adversaries \mathcal{A} , we have that:

$$\text{Adv}_\Sigma^{\text{EUF-CMA}}(\mathcal{A}) = \Pr \left[\text{Expt}_\Sigma^{\text{EUF-CMA}}(\mathcal{A}) \rightarrow 1 \right]$$

is a negligible function in κ , where $\text{Expt}_\Sigma^{\text{EUF-CMA}}(\mathcal{A})$ is defined in Figure 1.

$\text{Expt}_\Sigma^{\text{EUF-CMA}}(\mathcal{A})$:	$\text{Sign}^*(sk, m)$:
1. $(pk, sk) \leftarrow \Sigma.\text{KeyGen}(1^\kappa)$	1. $\mathcal{L} \cup m$
2. $\mathcal{L} = \emptyset$	2. $\sigma \leftarrow \text{Sign}^*(sk, m)$
3. $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\text{Sign}^*(sk, \cdot)}(pk)$	3. return (m, σ)
4. If $m^* \notin \mathcal{L} \wedge \text{Verify}(pk, m^*, \sigma^*) = 1$ return 1	
5. Else 0	

Figure 1. The EUF-CMA security experiments for DSAs.

2.3. Hash Functions

For the purposes of this paper, when we refer to a hash function, we are specifically referring to an *unkeyed compression* function, which takes strings of arbitrary length and outputs digests of some fixed length l . Furthermore, when we informally refer to a secure or good hash function, we mean a hash function that is *collision-resistant*, that is, it should be infeasible for an adversary to produce two different messages m_0, m_1 that evaluate to the same digest. We now formalize the notion of a family of collision-resistant of hash functions.

Definition 3. Let \mathcal{H} be a family of functions $\mathcal{H} = \cup_n \{H_i : \{0, 1\}^* \rightarrow \{0, 1\}^l\}_{i \in \mathcal{I}}$. We call this family of functions a family of collision-resistant families of hash functions if:

There is a PPT time algorithm to sample $H_i \in \mathcal{H}, \forall i \in \mathcal{I}$;

Given $m, i, H_i(m)$ can be computed in poly-time;

For all adversaries \mathcal{A} , and $\forall i \in \mathcal{I}$, we have that

$$\text{Adv}_{\mathcal{H}}^{\text{collision}}(\mathcal{A}) = \Pr \left[\text{Expt}_{\mathcal{H}}^{\text{collision}}(\mathcal{A}) \rightarrow 1 \right]$$

is a negligible function, where $\text{Expt}_{\mathcal{H}}^{\text{collision}}(\mathcal{A})$ is defined in Figure 2.

$\text{Expt}_{\mathcal{H}}^{\text{collision}}(\mathcal{A})$:
1. $H_i \leftarrow \mathcal{H}$.
2. $(m_0, m_1) \leftarrow \mathcal{A}^{H_i}$.
3. If $H_i(m_0) = H_i(m_1)$ return 1, else 0.

Figure 2. The collision-resistance security experiments for the hash function family \mathcal{H} .

3. The Dandelion Protocol

In this section of the paper, we outline the users and network settings of the Dandelion protocol [15], the protocol itself, the message flows, and the data structure.

3.1. Network and Users

The fundamental network structure of Dandelion is that of an asynchronous, open network, where users, or nodes, may join the network at any time, and may participate in any number of requests they choose. More precisely, by asynchronous we mean that there is no shared network clock, and that communication between clients and nodes can be responded to by the other at any point in time. This description also allows for situations where nodes may adaptively go on- or offline, including because of crashes, disconnections, or other reasons [15].

The network is parameterized by the following: a list of all users and all related information \mathbb{U} , an existentially unforgeable under chosen message attack (EUF-CMA-secure) digital signing algorithm (DSA) Σ , a collision-resistant hash function Hash, and a variable that denotes the maximum number of unclaimed transactions a user may have at once $|\max_{PJ}|$. We refer to a user's unclaimed transactions as the user's *penny jar*.

Users, U , in the Dandelion protocol are parameterized by the following:

Keys: A pair of Σ keys (pk_U, sk_U) , where sk remains private, and pk acts as part of U 's public account ID;

Shard ID: Shard_{id} , a parameter to denote which "shard" of the network the user belongs to. Shards are finite collections of users and are assigned to users as they join the network using a proprietary technology not discussed in this work. A specific users' shard ID is denoted as Shard_{id}^U ;

Balance: The user's account balance, denoted by Balance. When we refer to a specific users' balance, we will use the notation Balance^U ;

Transaction ID: The number of sent transactions the user has completed is denoted by TxID, which is updated incrementally as new transactions are completed. To refer to a specific i -th transaction number, we adopt the notation $\text{Tx}_{\#i}$. In cases where we wish to refer to the transaction of a specific user, we denote it as TxID^{U_A} , and likewise for $\text{Tx}_{\#i}^{U_A}$. We also use the notation $U_A.\text{TxID}$ to denote a U_A 's transaction ID according to the internal state of another node;

Status: Status represents what state the user is in from the view of a different node. Each user has three possible states: Locked, Unlocked, and PreAbort. When a node is Unlocked, the node is free to create new transaction requests to send to the network. When a node is Locked, the rest of the nodes will reject transaction requests from it until the status is changed to Unlocked. PreAbort is the state nodes are set to when their transaction requests have been denied. Once a transaction request has been denied, the node must send out an abort request to the network, to which their status is set to PreAbort. The node must then receive enough authorizations to abort the transaction and then send all authorizations to the network to become Unlocked. When we refer to a specific user's status, we will use the notation $U.\text{Status}$;

Penny Jar: PJ, a collection of non-redeemed transfers to the user, called *pennies*. This list is held by the other nodes of the network. Each penny in the penny jar is denoted by p^i . Each element is of the form $p^i = (\text{penny}, \text{block header})$. A formal description of a penny and block header are included in Figure 3. When we refer to a specific user's penny jar, we denote it as PJ^U . We note that different nodes may have differing penny jars for the same user U . To differentiate the source of the penny jar (penny) we denote it as $\text{PJ}_{U_i}(p_{U_i})$;

Blocks: The record of transactions which the user has participated in. We use Chain to denote the complete set of all transactions, or *blocks*, in a user's chain in sequential order, and B_i to refer to the i -th block of the user's chain. We will also adopt the notation of B_Ω to refer to the last block currently part of a user's chain. When we need to distinguish multiple users' chains (blocks) we use the notation B^U (B_i^U). A description of a block and the block header are provided below, in Figure 3. We note here an important difference between Dandelion and other traditional

blockchains: while other blockchains’ blocks are collections of global transactions, in the Dandelion protocol, each user has their own individual chain, where each block is a single transaction.

B_i of User U_A :	Block Header:	Penny:
<ul style="list-style-type: none"> • $T_{\#j}^{U_A}$ • Balance • Hash • $T_{\#k}^{U_B}$ • amount • $(pk_s, \text{Shard}_{id}^s)$ • $(pk_r, \text{Shard}_{id}^r)$ • aux 	<ul style="list-style-type: none"> • $T_{\#j}^{U_A}$ • Balance • Hash • PennyHash $H(\text{Penny})$	<ul style="list-style-type: none"> • $T_{\#l}^s$ • amount • $(pk_s, \text{Shard}_{id}^s)$ • $(pk_r, \text{Shard}_{id}^r)$
$\text{Hash} = H((\text{Shard}_{id}^s, pk_s, \text{TxID}^s, \text{amnt}, \text{Shard}_{id}^r, pk_r), B_{\Omega}^r, B_{\Omega}^s)$		

Figure 3. The structure of blocks, headers, and pennies in Dandelion. We let s denote information associated to a sender, while r denotes receiver’s information, l , if either j or k , depending on which of U_A or U_B is the sender.

We assume every node maintains a list of all the publicly available information above (i.e., all information barring the users’ secret key) for each other user that is efficiently searchable on inputs $(pk_U, \text{Shard}_{idU})$. We call this list the node list, \mathcal{U} . Whenever a node joins the network, we assume that they obtain a node list generated from \mathbb{U} at the time of joining. In the real-world setting, when a node wishes to join the network, they would use discovery channels to download a copy of other existing node lists (signed by the providing node), and collate these lists to create and publish their own node list. Moreover, in real-world applications, nodes would continuously update and publish their node list while participating in network validation to determine the approximate number of total online nodes. This is used so that nodes can determine whether a transaction has received enough responses and authorizations for finalization. To see why this is necessary, consider a transaction created by an adversary that is sent to one honest node and two corrupted nodes. The adversary can trivially create a scenario where a fake transaction has a consensus of approval from the responses *collected*. However, as the transaction is fake, it should not be possible for fabricated transactions to obtain majority approval. Thus, utilize each node’s node list to prevent this curating of biased responses.

3.2. Message Flows

We now briefly introduce Dandelion, a permissionless directed acyclic graph (DAG) blockchain protocol, and its message flows, with a series of figures. For readability considerations, we split the complete Dandelion protocol into three sub-protocols: transaction requests, abort requests, and collection requests. We then describe the sub-protocol with three figures: message descriptions, message flows, and how each message is handled. We provide the descriptions for how messages in each sub-protocol are handled in Appendix A.

We begin with the transaction request aspect, as described in Figures 4 and 5, and process according to Figures A1–A3. The abort sub-protocol is described in Figures 6 and 7, and process according to Figures A4–A6. Finally, the collection pennies sub-protocol is described with Figures 8 and 9, and process according to Figures A7–A9.

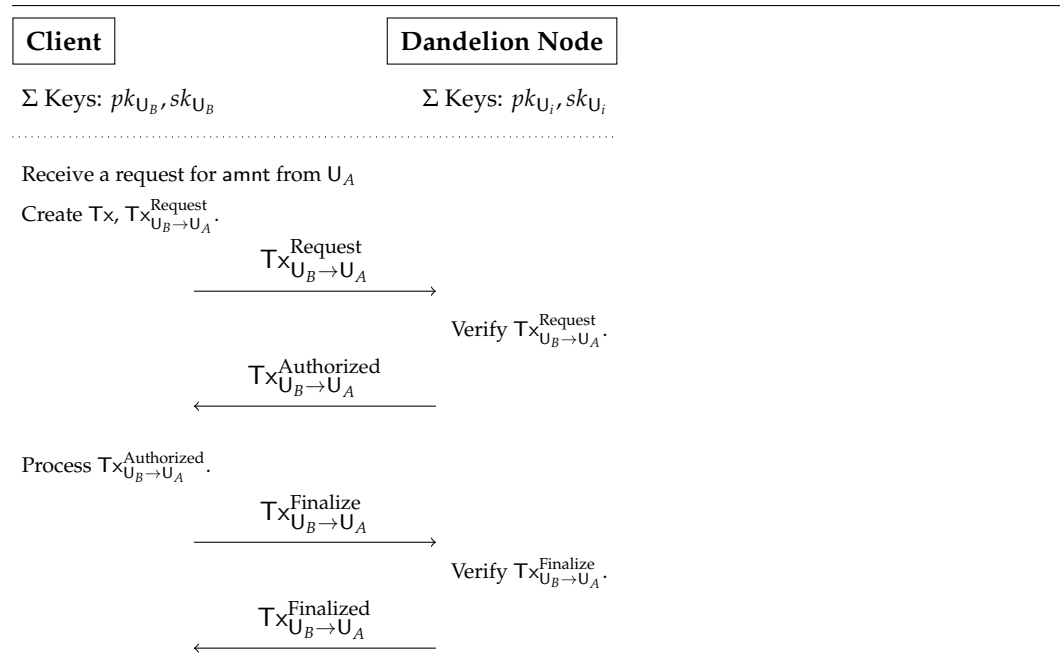


Figure 4. The Dandelion transaction request message flow.

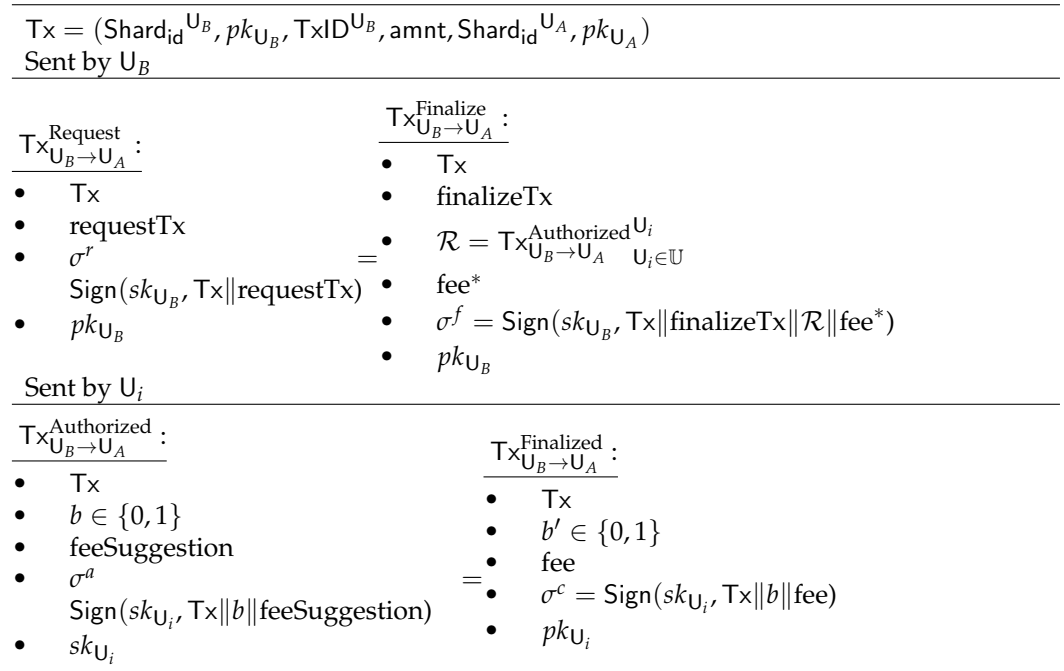


Figure 5. Dandelion transaction message description of senders and the network for a valid transaction T_x . We let the bit $b = 0$ denote denied, and $b = 1$ denote accepted, and fee is computed as the median of the lower two-thirds of responses.

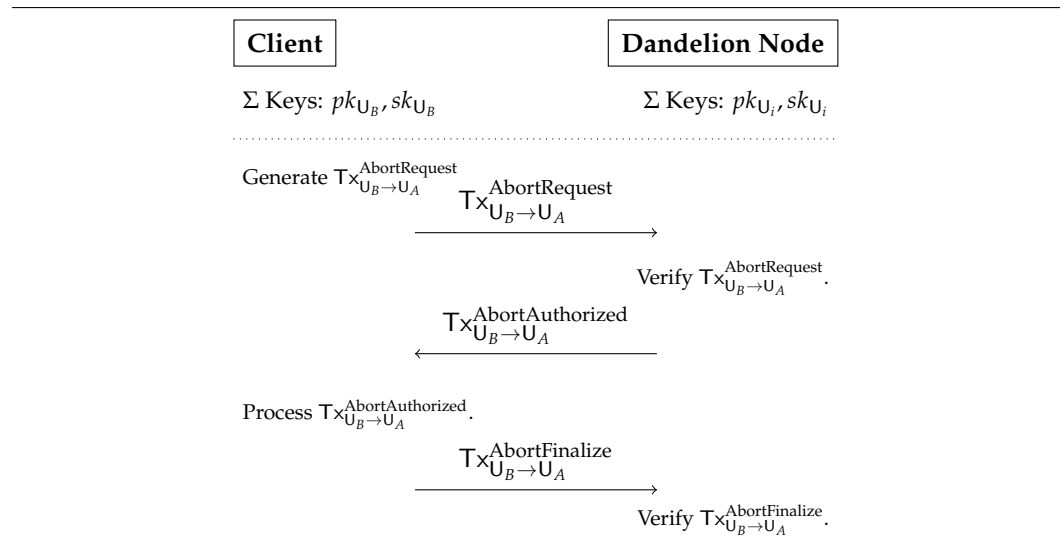


Figure 6. The Dandelion abort transaction message flow.

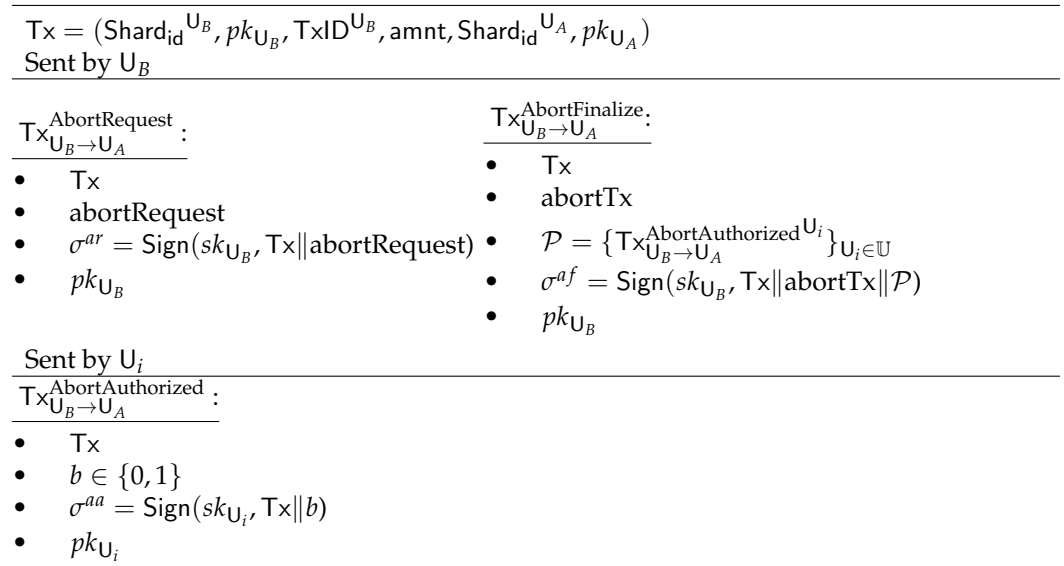


Figure 7. Dandelion abort message description of senders and the network for a valid transaction Tx. We let the bit $b = 0$ denote denied, and $b = 1$ denote accepted.

3.3. Data Structure

Unlike traditional blockchain protocols, which make use of linear chains and have potential new blocks compete for consensus to be added to the single chain each round, Dandelion uses the structure of a DAG. The DAG structure seeks to offer faster confirmation times and increased scalability than traditional linear blockchains without compromising security. In the last several years, there has been an increasing number of protocols built upon DAGs [4,16–20].

DAGs consist of a point set \mathcal{V} and an edge set \mathcal{E} . Each tuple (u, v) in the edge set represents a partial-order relationship between u and v , where order is defined as a directed path between u and v .

In the case of Dandelion, each element in the point set corresponds to a transaction Tx, and the directed path or order means that one transaction is linked to another. Furthermore, each user has their own DAG, and for every transaction there must exist a discretely matched pair of transactions from the sending account and the receiving account. Finally, Dandelion allows for these transaction blocks to be created asynchronously.

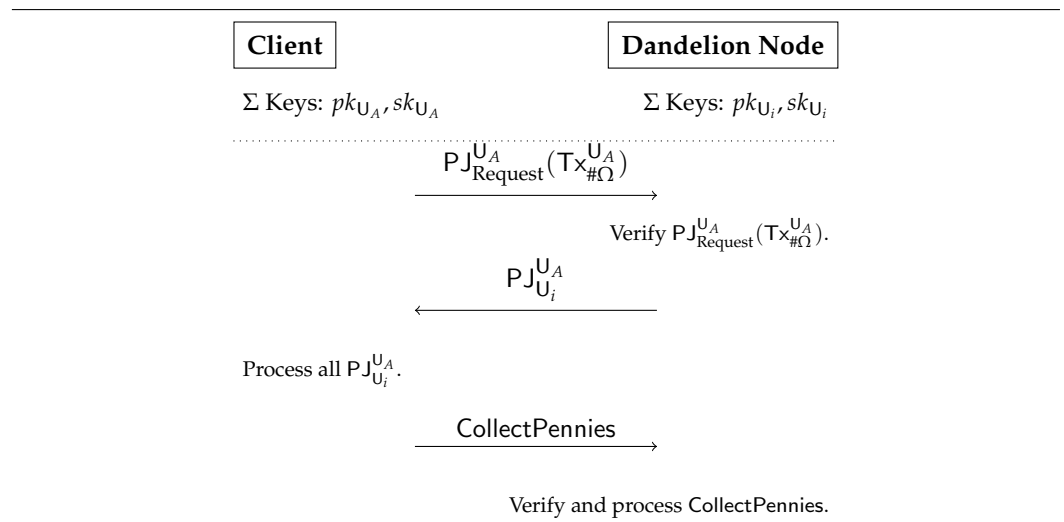


Figure 8. The Dandelion collect pennies protocol.

Sent by U_A	
$PJ_{Request}^{U_A}(Tx_{\#\Omega}^{U_A}) :$	CollectPennies :
<ul style="list-style-type: none"> $Tx_{\#\Omega}^{U_A}$ requestPennyJar $\sigma^{pr} = \text{Sign}(sk_{U_A}, Tx_{\#\Omega}^{U_A} \parallel \text{requestPennyJar})$ pk_{U_A} 	<ul style="list-style-type: none"> $\hat{\mathbb{P}}$ $Tx_{\#\Omega}^{U_A}$ $\sigma^{op} = \text{Sign}(sk_a, \hat{\mathbb{P}} \parallel Tx_{\#\Omega}^{U_A})$ pk_{U_i}
Sent by U_i	
$PJ_{U_i}^{U_A} :$	
<ul style="list-style-type: none"> PJ^{U_A} $\{\text{PennyHash}\}_{p \in PJ^{U_A}}$ $\sigma^{sp} = \text{Sign}(sk_{U_i}, PJ^{U_A} \parallel \{\text{PennyHash}\}_{p \in PJ^{U_A}})$ pk_{U_i} 	

Figure 9. Dandelion penny jar request description of senders and the network.

3.4. Comparisons

We now briefly highlight the unique structural features of Dandelion regarding other DAG-based protocols. We note here that a true comparison between Dandelion and other protocols is not truly possible on a structural level, and a performance comparison is outside the scope of this paper; the main goal of this paper is to prove the security of Dandelion in our model described in Section 4.

Unlike other peer-to-peer protocols, Dandelion operates on an individual level, and does not have a global or network-wide shared ledger of transactions. Instead, each account has its own chain recording each transaction they have participated in, which is stored on the network nodes. Furthermore, each account holder is responsible for advancing the state of their own chain, rather than a protocol determining when to add the next block to the shared chain.

A second fundamental difference is that network nodes do not communicate with one another to process requests. Dandelion uses a distinct consensus mechanism, as opposed to traditional consensus algorithms such as proof of work or proof of stake. Instead, Dandelion uses a so-called “client leader” model, where the account holder is responsible for collecting the responses from validator nodes. The account holder then forwards their collection of responses to the rest of the network, who individually confirm whether there is a two-thirds majority of approval before accepting and continuing.

Finally, as previously mentioned, Dandelion allows for completely asynchronous processing of payments and deposits. As a result of these design choices, a rigorous

comparison between the security of Dandelion and other well-known protocols would not be truly meaningful, and a performance comparison is not within the scope of this work.

4. Security Model

In this section, we outline the adversarial powers (via oracles) and security definitions developed for the Dandelion protocol. We begin with global network parameters needed for Dandelion, as well as several artificial parameters necessary for the proof of security.

4.1. Adversary Oracles

In this section of the paper, we present a series of oracles that are given to the adversary attempting to achieve some goal. We define the security criteria and the corresponding experiments following the descriptions of the oracles.

Before beginning the description of our oracles, we provide insight behind this choice of approach. Proofs of security for algorithms, such as DSAs, and (authenticated) key exchange protocols typically use oracles in security experiments, that are given to the adversary. This allows them to interact and control various aspects of the algorithm/protocol. In the case of key exchanges, the adversary can simulate the entire network, issuing messages on behalf of users, corrupt users, and more, effectively giving them near-total control while they attempt to win the security experiment. Our choice of using an oracle representation is to bring security proofs of blockchain protocols in line with other cryptographic algorithms through the use of oracles and formal security experiments.

To this end, we adopt a method similar to key exchange models, where the oracles are able to generate and process any message from the protocol. Thus, our oracles are defined to follow this approach, and provide the adversary oracles that are capable of generating and processing any message from the Dandelion protocol. We note here that our choice of model can be thought of as an oracle-based generalization of Garay et al.’s security model [21], which gives an adversary the ability to broadcast any message they wish, adaptively corrupt nodes, and change the source of any message. However, their model is not truly appropriate for our purposes, as Dandelion does not utilize rounds and a global ledger. Instead, each transaction is completed separately, and we make use of oracles which act on a transactional level.

We first define two additional lists that we require for our security analysis: \mathbb{C} , which is a list of users that have been corrupted by the adversary; \mathbb{T} , a list of non-active nodes; and \mathcal{L}_{Tx} , a list of transactions the adversary has created via oracle queries. Each of \mathbb{C} , \mathbb{T} , and \mathcal{L}_{Tx} begins empty and is updated adaptively in response to the actions of the adversary. Importantly, \mathbb{C} allows the adversary to view the corrupted node’s secret keys and to arbitrarily decide the node’s responses to transaction requests. Finally, we note that all transactions require $amnt > 0$, and all other values will result in a reject message \perp .

Add: Generates a new node U , under the control of \mathcal{A} . The oracle first checks whether $\frac{|\mathbb{C}|+1}{|\mathbb{U}\setminus\mathbb{T}|} < \frac{1}{3}$. If yes, then the node’s Σ keys are generated, along with its shard ID $Shard_{id}$; its balance and transaction are each set to 0, Status is set to Unlocked, and its chain Chain and penny jar PJ are set to empty. The node is added to both $(pk_U, sk_U, Shard_{id}^U, 0, 0, Unlocked, -, -) \cup \mathbb{U}$ and $(pk_U, Shard_{id}^U) \cup \mathbb{C}$, and (pk_U, sk_U) are returned to \mathcal{A} . If no, then the oracle returns \perp .

Corrupt $(pk_U, Shard_{id}^U)$: Corrupts a node to give \mathcal{A} the node’s secret key and control over it. The oracle first checks whether the node is in the master list of users \mathbb{U} and whether $\frac{|\mathbb{C}|+1}{|\mathbb{U}\setminus\mathbb{T}|} < \frac{1}{3}$. If yes, it then returns pk_U ’s corresponding secret sk_U and $(pk_U, sk_U, Shard_{id}^U) \cup \mathbb{C}$. If not, then the oracle returns \perp .

Down $(pk_U, Shard_{id}^U)$: Takes a node temporarily offline so that the node will miss the next transaction or collection request, simulating the node missing the next round of network validation. The oracle first checks whether the node is in the master list of users \mathbb{U} and whether $\frac{|\mathbb{C}|+1}{|\mathbb{U}\setminus\mathbb{T}|} < \frac{1}{3}$. If yes, then the oracle is successful and $(pk_U, Shard_{id}^U) \cup \mathbb{T}$, and it returns 1. Otherwise, the oracle fails and returns 0.

- BuildTx** $((pk_{U_A}, sk_{U_A}, \text{Shard}_{id}^{U_A}), (pk_{U_B}, \text{Shard}_{id}^{U_B}), \text{amnt})$: Creates a transaction request to send amnt from a corrupted node U_A to another node, U_B . First, the oracle confirms both that the user exists, and that $\text{amnt} \leq \text{Balance}^{U_A}$. If neither is true, then the oracle rejects. Otherwise, the oracle creates the corresponding Tx and $\text{Tx}_{U_A \rightarrow U_B}^{\text{Request}}$, and returns a $\text{Tx}_{U_A \rightarrow U_B}^{\text{Request}}$ message to the adversary and adds $\text{Tx} \cup \mathcal{L}_{\text{Tx}}$.
- TxAuthorize** $(\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}) \rightarrow \mathcal{R}$: Returns the responses of all nodes $U \in \mathbb{U} \setminus (\mathbb{T} \cup \mathbb{C})$ to the transaction request. First, the oracle does the following for each online non-corrupted node: it confirms that each of the users exist, that $U_B.\text{Status}$ is Unlocked, that $\text{amnt} \leq \text{Balance}^{U_B}$, that $|\text{PJ}^{U_A}| < |\text{max}_{\text{PJ}}|$, and it verifies the signature. The node then updates its node list. If a node accepts the transaction request as valid, it will return a $\text{Tx}_{U_B \rightarrow U_A}^{\text{Authorized}}$ message (see Figure 5) with $b = 1$, and will set $U_B.\text{Status} = \text{Locked}$. Otherwise, the node sets $b = 0$ and sends $\text{Tx}_{U_B \rightarrow U_A}^{\text{Authorized}}$. We describe how this oracle handles nodes that were previously offline in Appendix B.
- TxFinalize** $(\mathcal{R}, \mathcal{R}^A)$: Creates the message $\text{Tx}_{U_B \rightarrow U_A}^{\text{Finalize}}$ (see Figure 5) with the list of responses $\mathcal{R} \cup \mathcal{R}^A$, computes the final processing fee from the suggested fees by ordering the suggested fees in increasing order, and then takes the median of the first two thirds. It then returns the completed message to the adversary.
- TxComplete** $(\text{Tx}_{U_B \rightarrow U_A}^{\text{Finalize}})$: Completes the transaction. Each node processes $\text{Tx}_{U_B \rightarrow U_A}^{\text{Finalize}}$ by first checking if each response in $\mathcal{R} \cup \mathcal{R}^A$ is from an existing node by checking its own node list. If not, then that response is discarded. Next, they verify the signatures of the nodes they know of, and confirms if over two-thirds of the responses, relative to their node list, in $\mathcal{R} \cup \mathcal{R}^A$, accepted the transaction. If yes, then the node computes the next block B in U_B 's chain, updates U_B 's balance by subtracting the amount request and the fee calculated, updating the transaction ID, sets $U_B.\text{Status}$ to Unlocked, and adds the transaction to U_A 's penny jar to be collected later. Finally, it returns each U_i 's response to the adversary $\text{Tx}_{U_B \rightarrow U_A}^{\text{Finalized}^{U_i}}$, $b = 0$ if they are rejected and $b = 1$ if accepted. \mathbb{T} is emptied.
- AbortTx** (Tx) : Creates an abort transaction request for transaction Tx. The oracle first checks whether $\text{Tx} \in \mathcal{L}_{\text{Tx}}$; if not, then the oracle returns rejectedCause . Otherwise, the oracle computes and returns $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortRequest}}$; see Figure 7.
- AbortConfirmation** $(\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortRequest}})$: Returns the responses of all nodes $U \in \mathbb{U} \setminus (\mathbb{T} \cup \mathbb{C})$ to the abort transaction requests. Each node reviews the transaction Tx and checks whether the transaction is still in U_A 's penny jar and not in U_B 's chain, and verifies the signature. If true, then the node sets $b = 1$, and U_i creates the response $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$ (see Figure 7) with $b = 1$. Otherwise, it sets $b = 0$, and creates $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$ accordingly. All the responses are collected in the set \mathcal{P} and returned to the adversary.
- AbortFinalize** $(\mathcal{P}, \mathcal{P}^A)$: Creates the abort finalize message $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ (see Figure 7) under response set $\mathcal{P} \cup \mathcal{P}^A$, where \mathcal{P}^A are the responses the adversary inputs themselves for the corrupted nodes they control, and returns the completed message to the adversary. $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ is then returned.
- AbortComplete** $(\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortFinalize}})$: Completes the abort request. Each node is given a copy of $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$, and first checks if each response in $\mathcal{P} \cup \mathcal{P}^A$ is from an existing node by checking its own node list. If not, then that response is discarded. Next, they verify the signatures of the nodes they know of and confirm if over two-thirds of the responses, relative to their node list in $\mathcal{P} \cup \mathcal{P}^A$, accepted the abort. If U_B accepts, then the transaction is aborted, removed from node U_A 's penny jar PJ^{U_A} , $U_B.\text{Status}$ is set to Unlocked and the balance is updated, and the message $b' = 1$ is outputted by the node. Otherwise, the node U_i does nothing but output $b' = 0$. The collection of responses is outputted to the adversary.

- RequestPennyJar**($pk_{U_A}, Tx_{\# \Omega}^{U_A}$): Creates a penny jar request for node U_A . The oracle first checks whether the node exists (if not, then the oracle returns `rejectedCause`) and if $Tx_{\# \Omega}^{U_A}$ is U_A 's transaction ID on $B_{\Omega}^{U_A}$. Otherwise, the oracle returns to the adversary $PJ_{Request}^{U_A}$; see Figure 9.
- SendPennyJar**($PJ_{Request}^{U_A}$): Returns the penny jar that every available node has for node U_A . Each node first attempts to verify the signature. If the signature is valid, it outputs $PJ_{U_i}^{U_A}$; see Figure 9. Additionally, the node sends the corresponding PennyHash for each penny. Otherwise, it returns `rejectedCause`. The responses are collected in the collection \mathbb{P} and returned to the adversary.
- OrderPennies**(\mathbb{P}): Returns U_A 's ordered list of pennies according to the node $\hat{\mathbb{P}}$ from all collected penny jars, along with the corresponding hash of the penny. The order is determined by the U_A s. The oracle then returns `CollectPennies` to the adversary; see Figure 9.
- CollectPennies**($\hat{\mathbb{P}}, \sigma^{op}, pk_{U_A}$): Sends the ordered list of pennies to the other nodes to redeem all pennies. First, it verifies the signature σ^{op} . If the signature verifies, then U_i updates U_A 's chain by hashing U_A 's preceding block along with the corresponding block from the sender's chain for the penny (using the blocker header to find the correct block) and creating the next block for U_A . This is repeated for each penny, and during each iteration, U_A 's balance and transaction ID are updated appropriately. Then, U_i empties U_A 's penny jar PJ^{U_A} . We describe how this oracle handles nodes that were previously offline in Appendix B.
- TxHistory**($pk_{U_A}, Shard_{id}^{U_A}, U_A, i$): Returns the i -th transaction Tx for node ($pk_{U_A}, Shard_{id}^{U_A}$) by searching the node's chain. If there is no such transaction, the oracle returns \perp .
- BlockHistory**($pk_{U_A}, Shard_{id}^{U_A}, U_A, i$): Returns the i -th block in node U_A 's chain. If there is no such block, then the oracle returns \perp .
- Hash**(-): Returns $H(m)$.

4.2. Security

In addition to resistance to double-spending attacks, we now provide a list of security properties for the Dandelion protocol. We first provide an informal description of the various properties that have been, we believe, necessary for practical use for the Dandelion protocol, in addition to resistance to double-spending attacks. We then capture these with notions in a formal setting, along with security experiments in which the adversary is given access to the above oracles and must achieve some related goal.

- Correctness:** If $|\max_{pj}| = \infty$, then all requests that were honestly created will be authorized by honest nodes in the network.
- Fault Tolerance:** An adversary, with some number of corrupted nodes, should not have monopolistic control over the approval of transactions or the processing of abort transaction requests.
- Decidability:** All transaction requests must either be completed or rejected.
- Unforgeable Transactions:** An adversary, with some number of corrupted nodes, cannot issue transaction requests that can become finalized on behalf of non-corrupted nodes.
- Non-repudiation:** An adversary, with some number of corrupted nodes, cannot issue abort transaction requests that are able to become finalized on behalf of non-corrupted nodes.
- Unforgeable Collection:** An adversary, with some number of corrupted nodes, cannot issue a request for the penny jar of honest nodes that causes honest nodes to return the corresponding penny jar.
- Immutability:** An adversary, with some number of corrupted nodes, cannot modify an existing users' chain.

Quantum Resistance: The above properties must hold in the presence of an adversary with quantum computing capabilities.

We call the first three properties the *functionality* properties, as they ensure a basic level of the usability of the Dandelion protocol. Correctness ensures that honest attempts by potential users should be accepted by others executing the protocol correctly. An honest transaction should only be rejected if the receiver’s penny jar is full. Thus, we must prove that if the penny jar limit was infinite and these messages were well-formed and valid, then they will always be accepted.

Fault tolerance is a necessary property for basic functionality, as Dandelion is to be a distributed decentralized network, where malicious nodes may attempt to collude and influence the network at large. As such, we must determine the fault-tolerance threshold of Dandelion, the point at which the network is effectively controlled by adversaries.

The final functionality property is that of decidability. Each request must be processed by the other nodes in the network before responding. If there existed a transaction that could not be decided upon, then, by the definition of Dandelion, the requester’s account would become locked. Moreover, the requester must then create an abort transaction message to unlock the node. Thus, we must prove that all transactions are either accepted, rejected, or can be aborted successfully.

The remaining five properties will be referred to as the *security properties* of Dandelion. We will use the traditional security experiment framework of **Setup**, **Query**, **Challenge**, and **Finalize** to formalize the first four security properties. Regarding quantum resistance, we will make it explicit that we will be considering a probabilistic poly-time adversary with quantum-computing capabilities, and that are capable of performing Hash queries in superposition. While the following security experiments and the subsequent proofs can be downgraded to consider classic adversaries, we consider it prudent to focus on quantum-capable adversaries as the default, given the impending standardization of quantum-resistant algorithms and the increasing probability of scaleable quantum computers during the intended lifetime of Dandelion.

We first begin with the security experiment for unforgeable transaction requests.

Definition 4 (Unforgeability of Transactions). *Let κ be a security parameter, C be the challenger of this experiment, Σ be a signature scheme, and H a hash function, and let \mathcal{A} be an adversary interacting with Dandelion via the queries defined in Section 4.1 within the following security experiment $\text{Exp}_{\text{Dandelion}}^{\text{UnFTx}}(\mathcal{A})$:*

Setup. *The challenger generates all the public and private information of all users in the Dandelion network according to the Dandelion protocol. This includes generating the public and secret key for Σ with security parameter κ . The challenger then simulates the network operating by uniformly creating, at random, interactions between nodes (i.e., simulating transaction requests, processing transactions, collecting pennies on arbitrary nodes, etc). After many polynomial transactions, the challenger outputs all public information to the adversary \mathcal{A} and initializes $\mathbb{C} = \mathbb{T} = \mathcal{L}_{\text{Tx}} = \emptyset$;*

Query. *The adversary \mathcal{A} receives the generated public information and may query any of the oracles: Add, Corrupt, Down, BuildTx, TxAuthorize, TxFinalize, TxComplete, AbortTx, AbortConfirmation, AbortFinalize, AbortComplete, RequestPennyJar, SendPennyJar, CollectPennies, TxHistory, Hash, BlockHistory.*

Challenge. *At some point, \mathcal{A} stops and requests a challenge from C . C then uniformly and at random selects a node, U^* , from $\mathbb{U} \setminus (\mathbb{C} \cup \mathbb{T})$, whose status is set to unlocked, and gives \mathcal{A} the public information of that node. If no such nodes exist, then C selects a node at random and generates the transaction abort request on behalf of the node. It then processes this request on behalf of all honest nodes. The adversary is then given back access to their oracles, but is forbidden from corrupting U^* ;*

Finalize. *The adversary eventually stops and outputs a transaction $\text{Tx}^* \notin \mathcal{L}_{\text{Tx}}$, a request message $\text{Tx}_{\mathbb{U}_B \rightarrow \mathbb{U}_A}^{\text{Request}}$, and a response list $\mathcal{R}^{\mathcal{A}}$ for any number of nodes in \mathbb{C} . C accepts these inputs and*

computes the complete Dandelion transaction request protocol on \mathbb{T}_{x^*} and uses \mathcal{R}^A as well as the response of honest nodes to compute $\mathbb{T}_{x_{U_B \rightarrow U_A}}^{Finalize}$. After completing, the protocol \mathcal{C} returns 1 if the transaction was accepted and finalized, and 0 otherwise.

We say that \mathcal{A} wins the game if \mathcal{C} returns 1 and loses otherwise. We say Dandelion provides UnFTx-security if, for all adversaries, \mathcal{A} , the advantage function below is negligible in the security parameter:

$$\text{Adv}_{\text{Dandelion}}^{\text{UnFTx}}(\mathcal{A}) = \left| \Pr \left[\text{Expt}_{\text{Dandelion}}^{\text{UnFTx}}(\mathcal{A}) \rightarrow 1 \right] \right|.$$

We note here that the remaining security experiments have identical **Setup** and **Query** phases to Definition 4, and we will omit their description from the remaining definitions for space considerations.

Definition 5 (Non-repudiation of Transactions). Let κ be a security parameter, \mathcal{C} be the challenger of this experiment, Σ be a signature scheme, and H a hash function, and let \mathcal{A} be an adversary interacting with Dandelion via the queries defined in Section 4.1 within the following security experiment $\text{Expt}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A})$:

Challenge. At some point, \mathcal{A} stops and requests a challenge from \mathcal{C} . \mathcal{C} then uniformly and at random selects a node, U^* , from $\mathbb{U} \setminus (\mathbb{C} \cup \mathbb{T})$. If this node has a pending transaction \mathbb{T}_{x^*} that has not yet been finalized, the challenger then sends U^* 's public information, along with \mathbb{T}_{x^*} . Otherwise, if the node is set to unlocked, \mathcal{C} generates a random valid transaction request from U^* to another node, U' , in $\mathbb{U} \setminus (\mathbb{C} \cup \mathbb{T})$, and sends the request to all other honest nodes. The challenger then forwards U^* 's public information and the transaction \mathbb{T}_{x^*} to \mathcal{A} . The adversary is then given back access to their oracles, but is forbidden from corrupting U^* ;

Finalize. The adversary eventually stops and outputs an abort transaction request $\mathbb{T}_{x_{U_B \rightarrow U_A}}^{\text{AbortRequest}}$ and a response list \mathcal{P}^A for any number of nodes in \mathbb{C} . \mathcal{C} accepts these inputs and computes the complete Dandelion abort transaction request protocol on \mathbb{T}_{x^*} , $\mathbb{T}_{x_{U_B \rightarrow U_A}}^{\text{AbortRequest}}$, and uses \mathcal{P}^A as well as the response of honest nodes to compute $\mathbb{T}_{x_{U_B \rightarrow U_A}}^{\text{AbortFinalize}}$. After completing, the protocol \mathcal{C} returns 1 if the abort transaction request was accepted and finalized and 0 otherwise.

We say that \mathcal{A} wins the game if \mathcal{C} returns 1 and loses otherwise. We say Dandelion provides NonRep-security if, for all adversaries, \mathcal{A} , the advantage function below is negligible in the security parameter:

$$\text{Adv}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A}) = \left| \Pr \left[\text{Expt}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A}) \rightarrow 1 \right] \right|.$$

Definition 6 (Unforgeable Collection Transactions). Let κ be a security parameter, \mathcal{C} be the challenger of this experiment, Σ be a signature scheme, and H a hash function, and let \mathcal{A} be an adversary interacting with Dandelion via the queries defined in Section 4.1 within the following security experiment $\text{Expt}_{\text{Dandelion}}^{\text{UnCol}}(\mathcal{A})$:

Challenge. At some point, \mathcal{A} stops and requests a challenge from \mathcal{C} . \mathcal{C} then uniformly and at random selects a node, U^* , from $\mathbb{U} \setminus (\mathbb{C} \cup \mathbb{T})$ and confirms that U^* has uncollected pennies in their jar. If there are no such pennies, \mathcal{C} selects another node U' in $\mathbb{U} \setminus (\mathbb{C} \cup \mathbb{T})$ that is unlocked, and then generates a valid transaction from U' to U^* and completes the Dandelion transaction request protocol to add a penny to U^* 's penny jar. \mathcal{C} then sends U^* 's public information to \mathcal{A} , as well as the transaction ID from the last block in U^* 's chain, $\mathbb{T}_{x_{\# \Omega}}^{U^*}$. The adversary is then given back access to their oracles, but is forbidden from corrupting U^* ;

Finalize. The adversary eventually stops, then outputs $\text{PJ}_{\text{Request}}^{U^*}$. \mathcal{C} accepts the message and completes the Dandelion request penny jar protocol and returns 1 if the abort was accepted and finalized, and 0 otherwise.

We say that \mathcal{A} wins the game if \mathcal{C} returns 1 and loses otherwise. We say Dandelion provides UnCol-security if, for all adversaries, \mathcal{A} , the advantage function below is negligible in the security parameter:

$$\text{Adv}_{\text{Dandelion}}^{\text{UnCol}}(\mathcal{A}) = \left| \Pr \left[\text{Expt}_{\text{Dandelion}}^{\text{UnCol}}(\mathcal{A}) \rightarrow 1 \right] \right|.$$

Definition 7 (Immutability of Chains). Let κ be a security parameter, \mathcal{C} be the challenger of this experiment, Σ be a signature scheme, and H a hash function, and let \mathcal{A} be an adversary interacting with Dandelion via the queries defined in Section 4.1 within the following security experiment $\text{Expt}_{\text{Dandelion}}^{\text{Immutable}}(\mathcal{A})$:

Challenge. At some point, \mathcal{A} stops and selects a node U^* with a chain Chain of length $l > 1$. The adversary then submits the node U^* and its chain, Chain^* , to the challenger, along with a transaction, Tx^* , and index $1 \geq i \leq l - 1$;

Finalize. The challenger accepts $(U^*, \text{Chain}^*, \text{Tx}^*, i)$ and first confirms the node exists and that Chain^* is the current state of the user’s chain. If not, \mathcal{C} returns 0 and \mathcal{A} has lost the security experiment. Otherwise, \mathcal{C} then computes the hash $H(\text{Tx}^*, B^{U_i^*})$ and compares it with the Hash_{i+1} from block $i + 1$ of U^* ’s chain. If $H(\text{Tx}^*, B^{U_i^*}) = \text{hash}_{i+1}$, then \mathcal{C} returns 1, and 0 otherwise.

We say that \mathcal{A} wins the game if \mathcal{C} returns 1 and loses otherwise. We say Dandelion provides Immutable-security if, for all adversaries, \mathcal{A} , the advantage function below is negligible in the security parameter:

$$\text{Adv}_{\text{Dandelion}}^{\text{Immutable}}(\mathcal{A}) = \left| \Pr \left[\text{Expt}_{\text{Dandelion}}^{\text{Immutable}}(\mathcal{A}) \rightarrow 1 \right] \right|.$$

5. Proof of Security

In this section, we prove the security of the Dandelion protocol in our model. We first demonstrate that Dandelion satisfies the three functionality properties before moving on to the security properties and, finally, to double-spending attacks. We begin with correctness.

Theorem 1. Dandelion is correct.

Proof. We first assume that $|\text{max}_{\text{PJ}}| = \infty$; this represents receivers still having space available in their penny jar. Next, recall the following: the Dandelion protocol has two request messages, a transaction request, $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$, and an abort request, $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortRequest}}$. In both requests, the sender must provide the following information: $\text{Tx} = (\text{Shard}_{\text{id}}^{U_B}, pk_{U_B}, \text{TxID}^{U_B}, \text{amt}, \text{Shard}_{\text{id}}^{U_A}, pk_{U_A})$. Each request also contains a corresponding “requestTx” or “abortTx”, a signature over the transaction details, and one of the two messages. In either situation, an honest user will include valid details of the transaction, including public account information. Thus, nodes will receive valid inputs, confirm that the transaction details are valid, and verify the signature attached. That is, the nodes will confirm that both accounts exist, that the transaction ID is correct (or update it accordingly), and that the amount is no more than the sender’s balance. Since there is no limit to the number of transactions that the sender can have in their penny jar, the honest request would not be rejected. Thus, Dandelion is correct. \square

Theorem 2. The fault-tolerance threshold of Dandelion is $\frac{N}{3}$.

Proof. Recall that, in order for a transaction or an abort to be finalized, the nodes must first receive a $\text{Tx}_{U_B \rightarrow U_A}^{\text{Finalize}}$ or $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ message, containing a list of responses signed by other nodes in the network. The node confirms the existence of the other nodes and verifies the corresponding response and signature from said node. The request is finalized if over two-thirds of the responses in the request authorized the transaction. Each request is intended to be sent across the network to all active nodes, N . This implies that a transaction requires greater than $\frac{N}{3} + 1$ nodes to authorize the transaction. Now, let f denote the number of

nodes an adversary controls. If $f \geq \frac{N}{3} + 1$, then clearly the adversary has complete control over all requests. However, this is also true for $f \geq \frac{N}{3}$, as the adversary is able to act as a tie-breaker to authorize a transaction or to force users to abort transactions as they would be unable to reach a consensus from the network, giving the adversary effective control of the network instead of complete control. Thus, we have that the fault-tolerance threshold is $\frac{N}{3}$. Alternatively, we have that $3f < N$, where N is the number of active online nodes, and f is the number of nodes the adversary controls. \square

Corollary 1. *Dandelion is decidable.*

Proof. Let N denote the number of active online nodes in the network, and let \mathcal{A} be an adversary that controls $f < \frac{N}{3}$ nodes of the network. By Theorem 1, we have that Dandelion is correct, and so all requests that are honestly generated, and well-formed requests will be authorized by honest nodes, provided the receiver has space available in their penny jar. By the assumption that \mathcal{A} controls fewer nodes than the fault-tolerance threshold, and by Theorem 2, it does not have enough control over the network to solely decide which requests are authorized. Moreover, there does exist a sufficient number of honest nodes to either authorize valid transactions or reject invalid transactions. \square

We now establish the security properties of Dandelion through a series of security reductions.

Theorem 3. *Let Σ be a signature algorithm that is EUF-CMA-secure against quantum-computing-capable PPT adversaries, H be a hash function that is collision-free against quantum-computing-capable PPT adversaries, and \mathcal{A} be a quantum-computing-capable PPT adversary that controls less than $\frac{1}{3}$ of the Dandelion network. Then, except with negligible probability, the Dandelion protocol satisfies the unforgeability of transaction property.*

Proof. To prove the security against forged transaction requests, we will demonstrate that an adversary capable of doing so can be used as an oracle algorithm to win the EUF-CMA-security game (see Definition 2) against Σ . Let \mathcal{B} be a quantum-computing-capable adversary against the EUF-CMA-security of Σ . We then consider \mathcal{B} while it is in the EUF-CMA-security experiment, as described in Figure 1; that is, it is given a public key pk^* and oracle Sign^* programmed with the corresponding secret key sk^* . Now, assume that there exists a quantum-computing-capable PPT adversary \mathcal{A} that can win the $UnFTx$ -security of Dandelion by outputting a new transaction request $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$ and response list $\mathcal{R}^{\mathcal{A}}$ that results in the transaction being finalized with non-negligible probability.

\mathcal{B} is able to use \mathcal{A} to win the EUF-CMA-security game as follows. First it picks a suitable collision-free hash function H , initializes the following lists— $\mathcal{U}, \mathcal{T}, \mathcal{C}$, generates Σ keys for $N - 1$ other nodes, defines the distribution D , and uses pk^* as the public key for the N -th node. \mathcal{B} then simulates the Dandelion protocol for some polynomial amount of time. This includes generating shard IDs for nodes as it adds them to the list of nodes, assigning account balances, arbitrarily deciding on transactions between random users, and keeps track of all user data, both, private and public. It uses its Sign^* oracle to produce signatures on behalf of the account represented with public key pk^* . \mathcal{B} then simulates \mathcal{A} and the $\text{Expt}_{Dandelion}^{UnFTx}(\mathcal{A})$ and acts as the challenger. As \mathcal{B} generated all but one node’s public key, it is able to answer all but one of the oracle queries \mathcal{A} may make, including Add and Corrupt queries, ensuring \mathcal{A} does not surpass the fault-tolerance threshold. The only exception is for a Corrupt query on $(pk^*, \text{Shard}_{id_{pk^*}})$. Thus, the simulation is perfect to \mathcal{A} through both the **Setup** and **Query** phases until \mathcal{A} queries Corrupt on pk^* .

We let E denote the event that \mathcal{A} queries Corrupt on the account with public key pk^* , and let q_c and q_a denote the number of Corrupt and Add queries made, respectively, by \mathcal{A} . During the simulation of \mathcal{A} by \mathcal{B} , we define an abort and restart condition if E happens, and we wish to bound the probability of this. First, we know that it must hold that $q_c + q_a < \frac{N}{3}$. Thus, the maximum number of Corrupt queries \mathcal{A} may make is $\frac{N}{3} - 1$. We can then upper-bound the probability of E by using a hyper-geometric distribution.

There are N total nodes, $\frac{N}{3} - 1$ accounts are drawn, and there is only a single success that is pk^* . Thus, we have:

$$\Pr[E] = \binom{1}{1} \frac{\binom{N-1}{\frac{N}{3}-1}}{\binom{N}{\frac{N}{3}-1}} = \frac{1}{3} - \frac{1}{N}.$$

Therefore, we have an upper bound of $\Pr[E] < \frac{1}{3}$ for $N > 3$, and \mathcal{B} will be forced to restart its simulation with probability $\frac{1}{3}$. When E does not occur, then \mathcal{B} will select the node with public key pk^* as the challenge for \mathcal{A} , instead of a truly random non- \mathcal{A} -controlled node. \mathcal{B} continues the simulation until \mathcal{A} outputs a $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$. \mathcal{B} then parses the $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$ message and forwards $\text{Tx} \parallel \text{requestTx}, \sigma^r$ to their own challenger. By assumption, we have that \mathcal{A} can successfully produce a $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$ and response list that will be accepted by the Dandelion network, which includes a valid signature for a node whose secret keys it does not have knowledge of. Thus, if $\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}$ is accepted by the network, then \mathcal{B} will have provided a successful forgery under pk^* . We can then conclude that:

$$\text{Adv}_{\text{Dandelion}}^{\text{UnFTx}}(\mathcal{A}) \leq 3 \cdot \text{Adv}_{\Sigma}^{\text{EUF-CMA}}(\mathcal{B}),$$

where the factor of 3 is to account for the probability of \mathcal{A} querying Corrupt on \mathcal{B} 's challenge public key. \square

Theorem 4. *Let Σ be a signature algorithm that is EUF-CMA-secure against quantum-computing-capable PPT adversaries, H be a hash function that is collision-free against quantum-computing-capable PPT adversaries, and \mathcal{A} be a quantum-computing-capable PPT adversary that controls less than $\frac{1}{3}$ of the Dandelion network. Then, except with negligible probability, the Dandelion protocol satisfies the non-repudiation of transactions property.*

Proof. We employ a similar strategy to the one used in Theorem 3. We consider a quantum-computing-capable PPT adversary \mathcal{B} attempting to win the EUF-CMA-security experiment on the signature scheme Σ . We also assume that there exists a quantum-computing-capable PPT adversary \mathcal{A} that is able to win the $\text{Expt}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A})$ experiment, as in Definition 5, with non-negligible probability. We will demonstrate a reduction that \mathcal{B} may employ to use \mathcal{A} as an oracle algorithm to win their experiment.

We have \mathcal{B} set up a Dandelion network identically to the way described in Theorem 3. It generates a signing key and account information for N nodes, using its challenge public key, pk^* , as part of the network. It also generates the distribution D . It then simulates Dandelion for polynomial time, making arbitrary valid transactions between nodes before stopping and beginning to simulate \mathcal{A} in the *NonRep* security experiment over the simulated network.

This simulation is perfect up to \mathcal{A} querying Corrupt on pk^* , and the probability of this occurring is upper-bounded by $\frac{1}{3}$. If \mathcal{A} does not perform this query, and eventually requests a challenge, \mathcal{B} forwards the account information associated with pk^* to \mathcal{A} , along with a challenge transaction Tx^* (that \mathcal{B} may or may not need to generate itself). \mathcal{A} is given back access to its oracles and eventually outputs a $\text{Tx}_{U_B \rightarrow U_A}^{\text{AbortRequest}}$ message and response list $\mathcal{P}^{\mathcal{A}}$, which \mathcal{B} parses and forwards to its own challenger $\text{Tx}^* \parallel \text{abortRequest}, \sigma^{\text{ar}}$. Again, by assumption, \mathcal{A} is able to win the $\text{Expt}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A})$ with non-negligible probability, and so it must have produced a valid signature under a public key for which it does not know the secret key. Thus, if \mathcal{A} would win the $\text{Expt}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A})$ by having the network accept the abort request as legitimate with that output, then \mathcal{B} will win the EUF-CMA experiment.

We conclude that:

$$\text{Adv}_{\text{Dandelion}}^{\text{NonRep}}(\mathcal{A}) \leq 3 \cdot \text{Adv}_{\Sigma}^{\text{EUF-CMA}}(\mathcal{B}),$$

where the factor of 3 is to account for the probability of \mathcal{A} querying Corrupt on \mathcal{B} 's challenge public key. \square

Theorem 5. Let Σ be a signature algorithm that is EUF-CMA-secure against quantum-computing-capable PPT adversaries, H be a hash function that is collision-free against quantum-computing-capable PPT adversaries, and \mathcal{A} be a quantum-computing-capable PPT adversary that controls less than $\frac{1}{3}$ of the Dandelion network. Then, except with negligible probability, the Dandelion protocol satisfies the unforgeable collection of transactions property.

Proof. We consider a quantum-computing-capable PPT adversary \mathcal{B} against the EUF-CMA-security of Σ . We also assume that there exists another quantum-computing-capable PPT adversary \mathcal{A} that can win the $\text{Expt}_{\text{Dandelion}}^{\text{UnCol}}(\mathcal{A})$, as in Definition 5, with non-negligible probability. \mathcal{B} sets up a simulation of the Dandelion network and includes its challenge public key pk^* as one of the nodes in the network. It also generates the distribution D . It runs the simulation for polynomial time before stopping. \mathcal{B} then begins to simulate \mathcal{A} in the *UnCol*-security experiment.

Once again, the simulation is perfect until \mathcal{A} queries *Corrupt* on pk^* , which occurs with probability bounded above $\frac{1}{3}$. If \mathcal{A} does not perform this query, and eventually requests a challenge, \mathcal{B} forwards the account information associated with pk^* to \mathcal{A} , along with $\text{Tx}_{\# \Omega}^{U^*}$. \mathcal{B} may need to generate a transaction to add a penny to the corresponding node’s penny jar. \mathcal{A} is given back oracle access, and eventually outputs a *RequestPennyJar* message. \mathcal{B} then takes this message and submits it to its own challenger. By assumption, \mathcal{A} is able to win the *UnCol*-security experiment with a non-negligible advantage, and so it must produce a valid signature for a node it does not control. Thus, if \mathcal{A} had won in the *UnCol*-security experiment, \mathcal{B} would have submitted a successful forgery in the EUF-CMA-security experiment.

We conclude that:

$$\text{Adv}_{\text{Dandelion}}^{\text{UnCol}}(\mathcal{A}) \leq 3 \cdot \text{Adv}_{\Sigma}^{\text{EUF-CMA}}(\mathcal{B}),$$

where the factor of 3 is to account for the probability of \mathcal{A} querying *Corrupt* on \mathcal{B} ’s challenge public key. \square

Theorem 6. Let Σ be a signature algorithm that is EUF-CMA-secure against quantum-computing-capable PPT adversaries, H be a hash function that is collision-free against quantum-computing-capable PPT adversaries, and \mathcal{A} be a quantum-computing-capable PPT adversary that controls less than $\frac{1}{3}$ of the Dandelion network. Then, except with negligible probability, the Dandelion protocol satisfies the immutability of chains property.

Proof. We once again use a similar approach to those used in the previous Theorems, except we do not consider a quantum-computing-capable PPT adversary, \mathcal{B} , against the EUF-CMA-security of Σ , but against the collision-free security experiment in Figure 3. We also assume that there exists a quantum-computing-capable PPT adversary, \mathcal{A} , that can win $\text{Expt}_{\text{Dandelion}}^{\text{Immutable}}(\mathcal{A})$ -security experiment in Definition 7.

As before, \mathcal{B} simulates the Dandelion network, but this time it generates all nodes’ signing keys and distributions D . This means that there is no *Corrupt* query that the adversary can make that will cause \mathcal{B} to restart the simulation. As a result, \mathcal{B} can perfectly simulate the *Immutable*-security experiment by using its own hash oracle to answer hash queries, and answer all other queries itself. Eventually, \mathcal{A} submits a node U^* , its chain Chain^* , an index i , and transaction Tx^* to \mathcal{B} . \mathcal{B} then parses the input and does the following: \mathcal{B} searches U^* ’s chain for block i , B_i , and the transaction information from block $i + 1$ for the transaction:

$$\tilde{\text{Tx}} = (\text{Shard}_{\text{id}}^{\text{send}}, pk_{\text{send}}, \text{TxID}^{U^*}, \text{amnt}, \text{Shard}_{\text{id}}^{\text{receive}}, pk_{\text{receive}}).$$

It then submits the following messages to its own oracle— $m_0 = \text{Tx}^*, B_i$, and $m_1 = \tilde{\text{Tx}}, B_i$. By assumption, \mathcal{A} wins the real *Immutable*-security experiment with non-negligible probability by producing a collision between $H(\text{Tx}^*, B_i)$ and the hash value used in the next block. We can, then, conclude that:

$$\text{Adv}_{\text{Dandelion}}^{\text{Immutable}}(\mathcal{A}) \leq \text{Adv}_{\mathcal{H}}^{\text{collision}}(\mathcal{B}).$$

Thus, if \mathcal{A} has successfully produced a collision, then \mathcal{B} will also have submitted a collision as well. \square

We now, finally, cover the double-spend attack.

Theorem 7. *Malicious parties cannot double-spend in Dandelion.*

Proof. Let \mathcal{B} denote a quantum-computing-capable PPT adversary that controls less than $\frac{1}{3}$ of the Dandelion network, Balance^* be the balance of a single account that \mathcal{B} tries to double-spend from, and Tx_1 and Tx_2 denote the two transactions attempting to double-spend.

Then, without loss of generality, if \mathcal{B} attempts to collect authorizations for Tx_1 first, it sends the corresponding (valid) transaction request $\text{Tx}_{\text{U}_B \rightarrow \text{U}_{A1}}^{\text{Request}}$. Any honest nodes that receive the request will set $\mathcal{B}.\text{Status}$ to Locked and, thus, will not authorize the transaction request $\text{Tx}_{\text{U}_B \rightarrow \text{U}_{A2}}^{\text{Request}}$ for Tx_2 . Likewise, any honest nodes that receive $\text{Tx}_{\text{U}_B \rightarrow \text{U}_{A2}}^{\text{Request}}$ first will set $\mathcal{B}.\text{Status}$ to Locked and, thus, will not authorize the transaction request $\text{Tx}_{\text{U}_B \rightarrow \text{U}_{A1}}^{\text{Request}}$.

Thus, \mathcal{B} will have partitioned the network into two sets, M_1 and M_2 , based on which request the node received first. In order for both transactions to be authorized, and successfully double-spend, it must be that $M_1, M_2 > \frac{2N}{3}$. However, $M_1 + M_2 = N$ and so only one of $M_i > \frac{2N}{3}$ where $i = 1$ or 2 . Consequently, \mathcal{B} cannot authorize both transaction simultaneously. \square

6. Conclusions

In this work, we develop an oracle-based security model for the peer-to-peer blockchain transaction protocol Dandelion.

Our framework provides an oracle model for adversaries to interact with the Dandelion network, allowing for adaptive corruptions, network additions, and take-downs of nodes in the network to simulate a dynamic and asynchronous network. We then define formal security experiments for the ideas of unforgeability, non-repudiation, and immutability for the Dandelion protocol.

As the main result of our paper, we prove both the functional and security criteria of the Dandelion protocol with tight security reductions to the security of its digital signature algorithm and hash function. We also prove resistance to double-spend attacks. In these proofs, we explicitly consider quantum-capable PPT adversaries in order to reflect potential future attacks against Dandelion as quantum-computing technology becomes more robust, and such attacks become more likely.

Author Contributions: Conceptualization, B.G. and A.M.; methodology, B.G. and A.M.; formal analysis, B.G. and A.M.; resources, A.M.; writing—original draft preparation, B.G.; writing—review and editing, B.G. and A.M.; visualization, B.G.; supervision, A.M.; funding acquisition, A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Natural Sciences and Engineering Research Council (NSERC) of Canada, Discovery Grant RGPIN-2019-06150, and Mathematics of Information Technology and Complex Systems (MITACS) grant number IT20420.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank Dandelion Networks Inc. for the opportunity to review and provide the proofs of security for the Dandelion protocol. The authors would like to particularly thank Paul Chafe for the numerous discussions on the technical details of the Dandelion protocol included in this manuscript.

Conflicts of Interest: The authors declare no conflict of interest. The funding agencies had no role in the design of the study, in the analyses, in the writing of the manuscript, or in the decision to publish the results.

Appendix A

In this appendix, we include a description of how each message in the Dandelion protocol is handled by either the client or a generic node in the network.

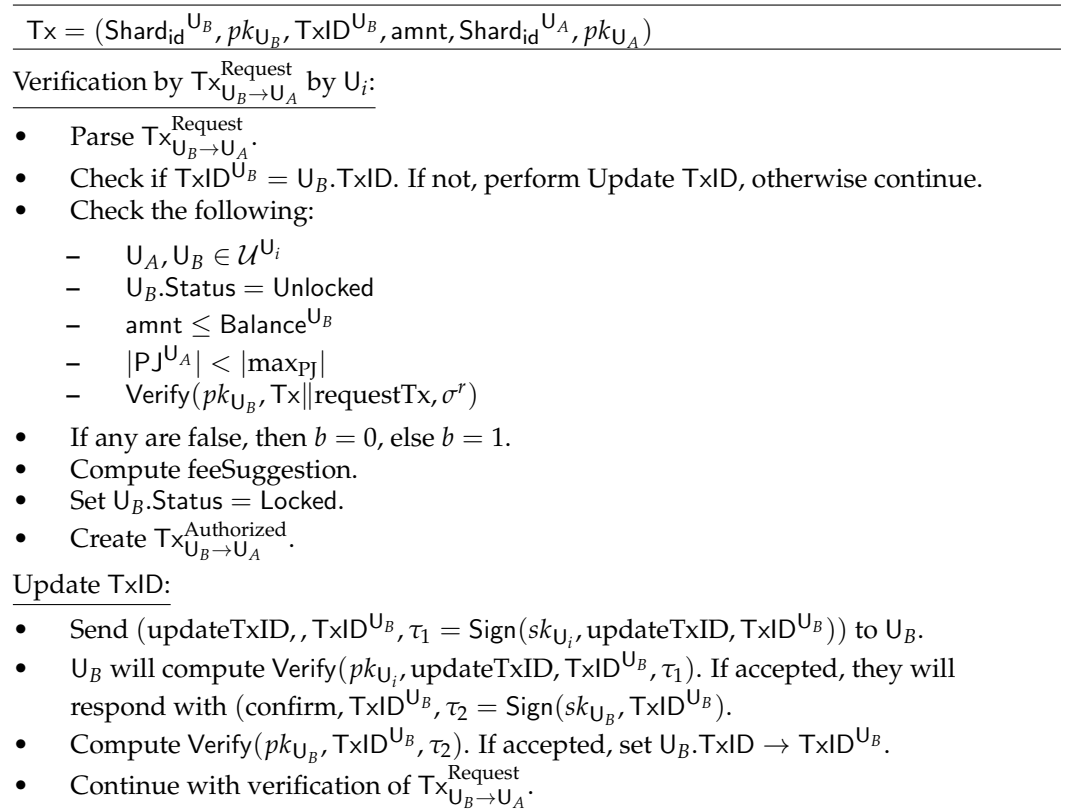


Figure A1. The verification of a transaction request $Tx_{U_B \rightarrow U_A}^{\text{Request}}$ by a Dandelion node U_i .

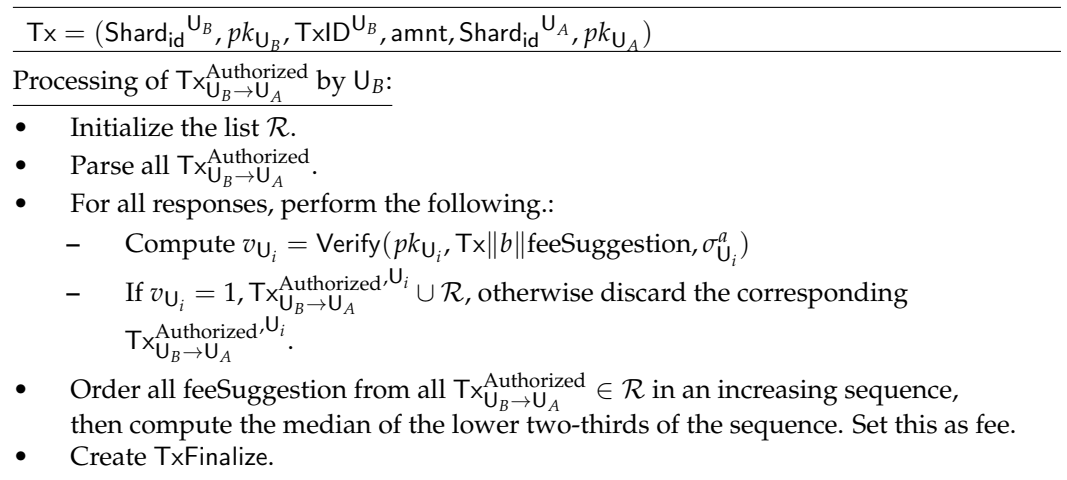


Figure A2. The process for how U_B responds to the authorization answers, $Tx_{U_B \rightarrow U_A}^{\text{Authorized}}$, from the network.

$Tx = (\text{Shard}_{id}^{U_B}, pk_{U_B}, TxID^{U_B}, \text{amnt}, \text{Shard}_{id}^{U_A}, pk_{U_A})$

Verification of TxFinalize by U_B :

- Parse TxFinalize and all $Tx_{U_B \rightarrow U_A}^{\text{Authorized}} \in \mathcal{R}$.
- For each response, determine if the corresponding came from a node in U_i 's node list. If a response is from an unknown node, discard it.
- Set $b' = 1$ if all of the following hold and 0 otherwise:
 - $\text{Verify}(pk_{U_B}, Tx || \text{finalizeTx} || \mathcal{R} || \text{fee}^*, \sigma^f) = 1$.
 - For all elements of \mathcal{R} , confirm that $Tx_{U_B \rightarrow U_A}^{\text{Authorized}}$ verifies.
 - Order all feeSuggestion from $Tx_{U_B \rightarrow U_A}^{\text{Authorized}} \in \mathcal{R}$ in an increasing sequence. Let fee' be the median of the lower two-thirds of the sequence of feeSuggestion. Check that $\text{fee}^* = \text{fee}'$.
 - Let B denote the number of $Tx_{U_B \rightarrow U_A}^{\text{Authorized}}$ where $b = 1$. Check that $B > \frac{|U|}{2}$.
- If $b' = 1$, then U_i aborts the transaction rolling back U_B 's chain and transaction ID, removing Tx from U_A 's penny jar, sets $U_B.\text{Status} = \text{Unlocked}$, and increasing $U_B.\text{Balance}$ by amnt.

Figure A3. The verification of TxFinalize by nodes in Dandelion.

$Tx = (\text{Shard}_{id}^{U_B}, pk_{U_B}, TxID^{U_B}, \text{amnt}, \text{Shard}_{id}^{U_A}, pk_{U_A})$

Verification by $Tx_{U_B \rightarrow U_A}^{\text{AbortRequest}}$ by U_i :

- Parse $Tx_{U_B \rightarrow U_A}^{\text{AbortRequest}}$.
- Set $b = 1$ if the following hold and 0 otherwise:
 - If $Tx \in PJ^{U_A}$ and that $\exists B \in \text{Chain}^{U_B}$ such that Tx is on that block.
 - Check that $\text{Verify}(pk_{U_B}, Tx, \text{abortRequest}) = 1$.
- Create $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$.

Figure A4. The verification of an abort request $Tx_{U_B \rightarrow U_A}^{\text{AbortRequest}}$ by a Dandelion node U_i .

$Tx = (\text{Shard}_{id}^{U_B}, pk_{U_B}, TxID^{U_B}, \text{amnt}, \text{Shard}_{id}^{U_A}, pk_{U_A})$

Processing of $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$ by U_B :

- Initialize the list \mathcal{P} .
- Parse all $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$.
- For all responses, perform the following.:
 - Compute $w_{U_i} = \text{Verify}(pk_{U_i}, Tx || \mu || b, \sigma_{U_i}^{ar})$
 - If $w_{U_i} = 1$, $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}.U_i} \cup \mathcal{P}$, otherwise discard the corresponding $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}.U_i}$.
- Create $Tx_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$.

Figure A5. The process for how U_B responds to the authorization answers, $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$, from the network.

$$Tx = (\text{Shard}_{id}^{U_B}, pk_{U_B}, TxID^{U_B}, \text{amnt}, \text{Shard}_{id}^{U_A}, pk_{U_A})$$

Verification of $Tx_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ by U_i :

- Parse $Tx_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ and all $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}} \in \mathcal{P}$.
 - For each response, determine if the corresponding came from a node in U_i 's node list. If a response is from an unknown node, discard it.
 - Accept and process the transaction abort if the following holds, otherwise the node does nothing:
 - $\text{Verify}(pk_{U_B}, Tx \parallel \text{abortTx} \parallel \mathcal{P} \parallel \mu^* \sigma^{af}) = 1$.
 - For all elements of confirm that \mathcal{P} , $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$ verifies.
 - Let B denote the number of $Tx_{U_B \rightarrow U_A}^{\text{AbortAuthorized}}$ where $b = 1$. Check if $B > \frac{|U|}{2}$.
 - If $b' = 1$, then U_i computes the next block in U_B 's chain according to the block structure, deducts the requested amount from U_B 's balance, updates $U_B.TxID$, adds the transaction to U_A 's penny jar, and sets $U_B.StatusUnlocked$.
 - Create $Tx_{U_B \rightarrow U_A}^{\text{Finalized}}$
-

Figure A6. The verification of $Tx_{U_B \rightarrow U_A}^{\text{AbortFinalize}}$ by nodes in Dandelion.

Verification by $PJ_{Request}^{U_A}$ by U_i :

- Parse $PJ_{Request}^{U_A}$. Check if all of the following hold:
 - Check whether $U_A \in U^{U_i}$.
 - Confirm that the $Tx_{\# \Omega}^{U_A}$ is the transaction ID of the last block in the user's chain.
 - Check that $\text{Verify}(pk_{U_A}, Tx_{\#j}^{U_A} \parallel \text{requestPennyJar}, \sigma^{pr}) = 1$.
 - If the above holds, create SendPennyJar . Otherwise, the node does nothing.
-

Figure A7. The verification of a collect penny request $PJ_{Request}^{U_A}$ by a Dandelion node U_i .

Ordering of Penny jars by $PJ_{Request}^{U_A}$:

- Initialize the list, \mathbb{P}
 - Parse each $PJ_{Request}^{U_A}$. Check if all of the following hold:
 - Check whether $U_A \in U^{U_i}$.
 - Check that $\text{Verify}(pk_{U_i}, PJ_{Request}^{U_A} \parallel \{\text{PennyHash}\}_{p \in PJ^{U_A}, \sigma^{sp}}) = 1$. If yes, then add $(PJ^{U_A}, \text{PennyHash}) \cup \mathbb{P}$.
 - Then creates a new list $\hat{\mathbb{P}}$, that orders all the elements of \mathbb{P} according to U_A .
 - Create CollectPennies .
-

Figure A8. Ordering of penny jars $PJ_{Request}^{U_A}$ from U_i by U_A .

Verification of CollectPennies by U_i :

- Check $\text{Verify}(pk_{U_A}, \hat{\mathbb{P}} \parallel \text{Tx}_{\#\Omega}^{U_A}) = 1$. If yes, U_i then goes through the ordered penny jar, converting each penny to a block in U_A 's chain according to the block structure as in Figure 3, updating the balance of U_A accordingly.
- If U_i has verified the collect pennies message and while processing pennies encounters a transaction it does not have any records of, it then undergoes the "updateChain" subroutine below.
- U_i then removes the pennies from U_A 's penny jar.

updateChain:

- U_i recovers the most recent transaction ID it has for U_A , $\text{Tx}_{\#\Omega}^{U_A}$.
- Computes $\sigma^{uc} = \text{Sign}(sk_{U_i}, \text{updateChain} \parallel \text{Tx}_{\#\Omega}^{U_A})$, and sends U_A the following $(\text{updateChain} \parallel \text{Tx}_{\#\Omega}^{U_A}, \sigma^{uc})$.
- U_A collects these messages and does the following,
 - Initialize the list M .
 - For each response, check $\text{Verify}(pk_{U_i}, \text{updateChain} \parallel \text{Tx}_{\#\Omega}^{U_A}, \sigma^{uc}) = 1$. If yes, then $\text{Tx}_{\#\Omega}^{U_A} \cup M$, otherwise the response is discarded.
 - U_A computes $\mu = \min(M)$. It then creates the message, $(\text{requestChainUpdate} \parallel \mu)$, and signs it, $\sigma^{rc} = \text{Sign}(sk_{U_A}, \text{requestChainUpdate} \parallel \mu)$. They then send $(\text{requestChainUpdate} \parallel \mu, \sigma^{rc})$ to all up-to-date nodes, U'_i .
 - The up-to-date nodes then check that $\text{Verify}(pk_{U_A}, \text{requestChainUpdate} \parallel \mu) = 1$. If not, they do nothing. Otherwise, the node creates the list $\mathfrak{B} := \{\mathfrak{B}_i\}_{i>\mu}$. Then, it sends the following message to U_A : $(\mathfrak{B}, \sigma^b = \text{Sign}(sk_{U'_i}, \mathfrak{B}))$.
 - U_A then initialized the list \mathbb{B} , then for each response that $\text{Verify}(pk_{U'_i}, \mathfrak{B}, \sigma^b) = 1$. If yes, then U_A adds $(\mathfrak{B}, \sigma^b, pk_{U'_i})$ to \mathbb{B} . Otherwise, the response is discarded.
 - U_A then creates the message $(\mathbb{B}, \sigma^B = \text{Sign}(sk_{U_A}, \mathbb{B}))$, and sends the message to the nodes that send the "updateChain" message.
- U_i waits until it receives the message (\mathbb{B}, σ^B) from U_A . It then checks that $\text{Verify}(pk_{U_A}, \mathbb{B}) = 1$. If not, then U_i does nothing but discard the message. Otherwise, U_i continues.
- U_i updates its node list, then checks each response $(\mathfrak{B}, \sigma^b, pk_{U'_i})$ in \mathbb{B} to verify the signature. The node discards each response whose signature it cannot verify, and creates the list $\hat{\mathbb{B}}$ to add verified responses to.
- If $|\hat{\mathbb{B}}| > \frac{|\mathbb{U}|}{2}$, then U_i accepts the new blocks. It then processes each of these new blocks and adds them to U_A 's chain according to the block creation routine. This includes adding pennies to the jars of other users in the case of newly added blocks to U_A 's chain where U_A is listed as sender. It also removes the pennies from U_A 's ordered penny jar it did not have records for.
- Lastly, U_i then sets $U_A.\text{Status} = \text{Unlocked}$ if it is not already unlocked.

Figure A9. Verification of CollectPennies by U_i and updating U_A 's chain.

Appendix B

We now discuss how nodes, which are out of sync due to being offline, are caught up in our oracle model according to the Dandelion protocol. We note that nodes will only recognize that they have become out-of-sync when either a client begins collecting transaction authorizations, as the client's transaction ID will differ from the node's records, or when the client attempts to collect pennies which the node has no record of.

$\text{TxAuthorize}(\text{Tx}_{U_B \rightarrow U_A}^{\text{Request}}) \rightarrow \mathcal{R}$: If a node was previously offline and, thus, has a different TxID for U_B , the node undergoes the updating process by sending the client a signed request for confirmation of their current transaction ID message, “updateTxID”. In response to a verified message, they receive a signed message of U_B ’s current TxID. The responses are collected into a list \mathcal{R} and returned to A .

$\text{CollectPennies}(\hat{\mathbb{P}}, \sigma^{op}, pk_{U_A})$: If a node was offline for transactions where U_A was the recipient and does not have records of these transactions, then they must update their copy U_A ’s chain. To do so, the U_i and U_A do the following: U_i determines U_A ’s most recent transaction ID, $\text{Tx}_{\# \Omega}^{U_A}$, from the last block in the chain and sends it along with an update chain message that is signed. U_A collects the update chain messages and computes the minimum of all transaction IDs, μ . The client tries to verify the signature then sends an “requestChainUpdate” message to the other nodes along with μ that is signed, if successfully verified, and otherwise does nothing. The other nodes then attempt to verify the signature on the request, and if successful, these nodes return a signed message of all blocks $\{B_i\}_{i > \mu}$ currently recorded with a transaction ID greater than μ . U_A then attempts to verify each response, discarding ones it can not verify, before sending the list of responses as a signed message to the necessary nodes. These nodes update their node lists and attempt to verify U_A ’s signature over the message. If successful, they then attempt to verify the signature of each response while discarding those that fail. If U_i verifies over two-thirds of the response, relative to the size of their node list, they then add these blocks to U_A ’s chain and process each transaction according to the information recorded on the block. This includes adding pennies to the jars of nodes listed as the receiver’s among the new blocks, or removing pennies from U_A ’s jar, as the corresponding block is added. Finally, U_i sets $U_A.\text{Status} = \text{Unlocked}$.

References

1. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. *Decentralized Bus. Rev.* **2008**, *2008*, 21260.
2. Buterin, V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. Available online: <https://github.com/ethereum/wiki/wiki/White-Paper> (accessed on 23 February 2022).
3. Khosravi, H.; Kitto, K.; Williams, J.J. RiPPLE: A Crowdsourced Adaptive Platform for Recommendation of Learning Activities. *arXiv* **2019**, arXiv:1910.05522. [CrossRef]
4. Sompolinsky, Y.; Zohar, A. PHANTOM: A Scalable BlockDAG Protocol. *IACR Cryptol. ePrint Arch.* **2018**, *2018*, 104.
5. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; Association for Computing Machinery: New York, NY, USA, 2018. [CrossRef]
6. David, B.; Gaži, P.; Kiayias, A.; Russell, A. Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain. In *Advances in Cryptology—EUROCRYPT 2018*; Nielsen, J.B., Rijmen, V., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 66–98.
7. Deloitte. *Deloitte’s 2020 Global Blockchain Survey*; Deloitte: London, UK, 2020.
8. Yue, X.; Wang, H.; Jin, D.; Li, M.; Jiang, W. Healthcare Data Gateways: Found Healthcare Intelligence on Blockchain with Novel Privacy Risk Control. *J. Med. Syst.* **2016**, *40*, 1–8. [CrossRef] [PubMed]
9. Azaria, A.; Ekblaw, A.; Vieira, T.; Lippman, A. Medrec: Using blockchain for medical data access and permission management. In Proceedings of the 2016 2nd International Conference on Open and Big Data (OBD), Vienna, Austria, 22–24 August 2016; pp. 25–30.
10. Song, J.M.; Sung, J.; Park, T. *Applications of Blockchain to Improve Supply Chain Traceability*; Elsevier: Amsterdam, The Netherlands, 2019; Volume 162, pp. 119–122.
11. Al-Rakhami, M.S.; Al-Mashari, M. A Blockchain-Based Trust Model for the Internet of Things Supply Chain Management. *Sensors* **2021**, *21*, 1759. [CrossRef] [PubMed]
12. Liao, D.Y.; Wang, X. Design of a Blockchain-Based Lottery System for Smart Cities Applications. In Proceedings of the 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC), San Jose, CA, USA, 15–17 October 2017; pp. 275–282. [CrossRef]
13. Shor, P. Algorithms for quantum computation: discrete logarithms and factoring. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA, 20–22 November 1994; pp. 124–134. [CrossRef]

14. Grover, L.K. A Fast Quantum Mechanical Algorithm for Database Search. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, STOC '96, Philadelphia, PA, USA, 22–24 May 1996; Association for Computing Machinery : New York, NY, USA, 1996; pp. 212–219. [[CrossRef](#)]
15. Chafe, P.; Mashatan, A.; Munro, A.; Goncalves, B.; Cameron, D.; Xu, J. DANDELION WHITEPAPER (v.1). Available online: <https://www.dandelionnet.io/> (accessed on 23 February 2022).
16. Sompolinsky, Y.; Lewenberg, Y.; Zohar, A. SPECTRE : Serialization of Proof-of-Work Events: Confirming Transactions via Recursive Elections. 2017. Available online: <https://www.semanticscholar.org/paper/SPECTRE-%3A-Serialization-of-Proof-of-work-Events-%3A-Sompolinsky-Lewenberg/65f1613a4f1b015fc64608b787227de0549f4cec> (accessed on 23 February 2022).
17. Zhou, T.; Li, X.; Zhao, H. DLattice: A Permission-Less Blockchain Based on DPoS-BA-DAG Consensus for Data Tokenization. *IEEE Access* **2019**, *7*, 39273–39287. [[CrossRef](#)]
18. Chevalier, P.; Kaminski, B.; Hutchison, F.; Ma, Q.; Sharma, S.; Fackler, A.; Buchanan, W.J. Protocol for Asynchronous, Reliable, Secure and Efficient Consensus (PARSEC) Version 2.0. *arXiv* **2019**, arXiv:1907.11445. [[CrossRef](#)]
19. Gupta, H.; Ram, D. CDAG: A Serialized blockDAG for Permissioned Blockchain. *arXiv* **2019**, arXiv:1910.08547.
20. Yu, H.; Nikolic, I.; Hou, R.; Saxena, P. OHIE: Blockchain Scaling Made Simple. *arXiv* **2018**, arXiv:1811.12628. [[CrossRef](#)]
21. Garay, J.; Kiayias, A.; Leonardos, N. The Bitcoin Backbone Protocol: Analysis and Applications. In *Advances in Cryptology—EUROCRYPT 2015*; Oswald, E., Fischlin, M., Eds.; Springer: Berlin, Germany, 2015; pp. 281–310.