Breaking the Binding: Attacks on the Merkle Approach to Prove Liabilities and its Applications

Kexin Hu, Zhenfeng Zhang, Kaiven Guo

Laboratory of Trusted Computing and Information Assurance,
Institute of Software, Chinese Academy of Sciences, Beijing, China.
{hukexin,zfzhang,guokaiwen}@tca.iscas.ac.cn

Abstract. Proofs of liabilities are used for applications, function like banks or Bitcoin exchanges, to prove the sums of money in their dataset that they should owe. The Maxwell protocol, a cryptographic proof of liabilities scheme which relies on a data structure well known as the summation Merkle tree, utilizes a Merkle approach to prove liabilities in the decentralized setting, i.e., clients independently verify they are in the dataset with no trusted auditor. In this paper, we go into the Maxwell protocol and the summation Merkle tree. We formalize the Maxwell protocol and show it is not secure. We present an attack with which the proved liabilities using the Maxwell protocol are less than the actual value. This attack can have significant consequences: A Bitcoin exchange controlling a total of n client accounts can present valid liabilities proofs including only one account balance in its dataset. We suggest two improvements to the Maxwell protocol and the summation Merkle tree, and present a formal proof for the improvement that is closest in spirit to the Maxwell protocol. Moreover, we show the DAM scheme, a micropayment scheme of Zerocash which adopts the Maxwell protocol as a tool to disincentivize double/multiple spending, is vulnerable to an multi-spending attack. We show the Provisions scheme, which adopts the Maxwell protocol to extend its privacy-preserving proof of liabilities scheme, is also infected by a similar attack.

Keywords: summation Merkle tree, proof of liabilities, Maxwell protocol, Bitcoin exchanges, the DAM scheme, the Provisions scheme

1 Introduction

Currency exchanges, as a crucial part of the digital currency ecosystem, provide conversion between currencies, fast deposit and withdrawal of funds, and other easy-to-use services to clients of dencentralized cryptocurrencies [7,13,14,15,18]. Holding a large amount of money, exchanges are attractive to hackers to commit crimes. Besides, internal theft, technical mistake, and investment failure can also result in catastrophic losses with clients permanently losing their assets. One of the most notorious events in exchanges' history is the bankruptcy of the first and for a long time largest Bitcoin exchange – Mt.Gox, following the loss of over 500 million USD worth of bitcoins owned by its clients in 2014 [8].

While such losses can never be fully mitigated, it's sensible for clients to ask exchanges to provide proofs of liabilities, i.e., proofs showing the sums of money that

exchanges should owe and client deposits are included in the sums. When cooperating with proofs of reserves, which are proofs of the money that exchanges actually own, it shows the financial status of exchanges if they periodically announce proofs. Hence, clients can timely withdraw their deposits when it is necessary.

A cryptographic proof of liabilities in the decentralized setting was first proposed by Greg Maxwell in 2013. Known as the Maxwell protocol [24], it relies on a data structure called the summation Merkle tree which is well known by the Bitcoin community. On a summation Merkle tree, each node stores a balance and a hash. A leaf node consists of the balance of a distinct account in the exchange's dataset and a hash of this account information. An internal node stores a sum of its child node balances, and a hash of its child hashes and the sum. In a bottom-up order, the summation Merkle tree is filled up. The root node stores the sum of all leaf node balances, i.e., the liabilities of the exchange. In the Maxwell protocol, an exchange establishes a summation Merkle tree and publishes the root node. The liabilities proof for a client, whose account is in the dataset, is an authentication path of the leaf node associated with the client account. Each client independently checks the validity of the received proof by reconstructing a path and verifying the equality between the computed root node and the published root node.

Proofs of liabilities may also consist of an auditor, as done by exchanges like Uphold, OKCoin, Kraken, etc. A simple approach is to give the auditor all data needed and announce the audit result, as done by Uphold [19]. This can be improved by allowing clients to independently verify they were included in the dataset used by the auditor, as done by OKCoin [4] and Kraken [11]. In particular, Kraken's proof of liabilities is motivated by the Maxwell protocol, but it requires an auditor to check the established Merkle tree. The involved auditor must be trusted and competent. However, there is no universally trusted auditor, and an auditor may collude with exchanges or compromise confidential client information. For proofs of liabilities with no auditor, Vaultoro [20] publishes an account list with anonymous client IDs and allow its clients to verify their accounts in the list. In addition to the Maxwell protocol, another approach with no auditor, drawn from the Provisions [6], employs heavier cryptography like zero-knowledge proofs and Pedersen commitments to obtain a privacy-preserving proof of liabilities scheme.

Compared to the these approaches, the Maxwell protocol which relies on the summation Merkle tree has the advantages of *simplicity*, *efficiency*, *account-privacy* and *decentralization*. The data structure of the underlying summation Merkle tree is simple and clear. For a dataset of size n, it only incurs an $\mathcal{O}(\log n)$ computation overhead per-verification. This is an alluring property for typically lightweight client. To protect account privacy, a leaf node only includes a hash result of an account information. A unique nonce is included when computing a hash result, and hence a client cannot link the received leaf node to other's accounts. Also, a client cannot link two received leaf nodes from different proofs because the account order in leaf nodes is a random permutation of the account order in the dataset. The Maxwell protocol is a decentralized proof of liabilities scheme that requires no auditor. Thus, the weaknesses of introducing an auditor can be avoided.

Conventional wisdom has asserted that the Maxwell protocol and the summation Merkle tree are secure against malfeasance by exchanges. There were no known techniques by which an exchange could earn benefits by deviating from

the protocol. Because they were believed to be secure, they are used to design or extend schemes [3,6].

In this paper, we show that the conventional wisdom is wrong: the Maxwell protocol, as well as the summation Merkle tree, is not secure. We go into and formalize the Maxwell protocol and the summation Merkle tree. We present an attack that exchanges can trick their clients that their deposits have been summed up into the published liabilities, whereas it only contains a small portion of the total amounts. We show that the lower bound of successfully verified liabilities is the maximum of all account balances in the exchange's dataset. This lower bound shows that for an exchange controlling a total of n client accounts can present valid liabilities proofs including only one account balance in its dataset.

The key idea behind our attack is that an exchange may provide *inconsistent* proofs to the sibling leaf nodes. Because the integrity of the nodes on a received proof cannot be verified, the exchange can modify node balances in a proof without being discovered. Verifications, using a published root whose balance has been modified, will pass without breaking the collision resistance of hash functions. We suggest two improvements to the Maxwell protocol and the summation Merkle tree that can resist this attack, and we give a formal proof for the one that is closest in spirit to the Maxwell protocol.

Our attack has impact on several existing schemes. We show that the scheme in [3], which is a decentralized anonymous micropayments scheme based on Zerocash [18], suffers from multi-spending attacks due to the usage of the Maxwell protocol as a tool to disincentivize double/multiple spending. We show the Provisions scheme [6], which adopts the Maxwell protocol to extend its privacy-preserving proof of liabilities scheme in its full paper [5], is also infected by a similar attack.

In summary, the contributions of this work are:

- We formalize the Maxwell protocol and the summation Merkle tree.
- We present an attack that breaks the security goal of the Maxwell protocol.
- We suggest two improvements to repair this problem, and give a formal security proof for the one that is closest in spirit to the Maxwell protocol.
- We present a multi-spending attack against the DAM scheme which uses the Maxwell protocol to disincentivize double/multiple spending.
- We present an attack against the extended version of the Provisions scheme that breaks the security goal of its proof of liabilities scheme.

We formalize the Maxwell protocol in Sect. 2. Our attack is formalized in Sect. 3. together with the improvements. Sect. 4 and 5 show the attacks on the Provisions and the DAM schemes. Sect. 6 concludes this paper.

2 The Maxwell Protocol

In this section, we present an algorithm depicting the complete process of the Maxwell protocol for proving liabilities, as shown in Algorithm 1. A proof of liabilities is used when an exchange, for example, holding a number of client deposits, wants to prove to its clients that their deposits have been accumulated in the total liabilities of this exchange. If *every* client has completed the verification, as long as the exchange showing that it controls an equal amount of money as its

presented liabilities, even if all clients apply for a withdrawal *simultaneously*, the exchange will not have insufficient money to support these requests.

When making a proof of liabilities, an exchange first generates a list $\mathcal{L}$ of size $n = 2^s$, where s is a positive integer. Each row of $\mathcal{L}$ consists of a client account information which contains the account balance v, the login ID, a nonce uniquely generated with the client and some other auxiliary data.

Algorithm 1 The Maxwell protocol

Input: $n = 2^s$: number of client accounts; $\mathcal{L}$: all account information list; H: a collision resistant hash function

Output: A result of the verification for a proof of liabilities

1: **procedure** FORMTREE ▷ Done by the exchange
2: $\texttt{Nodes} \leftarrow \texttt{empty}$ ▷ Create an empty binary tree with $2n-1$ nodes
3: $\tilde{\mathcal{L}} \leftarrow \mathsf{RandPermutate}(\mathcal{L})$ ▷ Permutate the rows of the input list randomly
4: **for** $i = n, i < 2n, i \leftarrow i+1$ **do** ▷ For each leaf node
5: $\texttt{Nodes}[i].v \leftarrow \tilde{\mathcal{L}}[i].v$, $\texttt{Nodes}[i].h \leftarrow \mathsf{H}(\tilde{\mathcal{L}}[i].v||\tilde{\mathcal{L}}[i].\mathrm{ID}||\tilde{\mathcal{L}}[i].nonce)$
6: **end for**
7: **for** $j = n-1, j > 0, j \leftarrow j-1$ **do** ▷ For each internal node
8: $\texttt{Nodes}[j].v \leftarrow \texttt{Nodes}[2j].v + \texttt{Nodes}[2j+1].v$
9: $\texttt{Nodes}[j].h \leftarrow \mathsf{H}(\texttt{Nodes}[j].v||\texttt{Nodes}[2j].h||\texttt{Nodes}[2j+1].h)$
10: **end for**
11: **for** $i = 1, i < 2n, i \leftarrow i+1$ **do**
12: **if** i is odd **then** $\texttt{Nodes}[i].t = 1$ ▷ Identify a node's position that "1" indicates the node is a right child
13: **else** $\texttt{Nodes}[i].t = 0$ ▷ "0" indicates the node is a left child
14: **end if**
15: **end for**
16: Publish $\texttt{Nodes}[1]$
17: **end procedure**
18:
19: **procedure** VERIFYPROOF ▷ Done by clients
20: $\overline{\texttt{ANodes}} \leftarrow \mathsf{Receive}()$ ▷ Receive a liabilities proof from the exchange
21: $\texttt{Node}' \leftarrow \mathsf{GetAccountInfo}()$ ▷ Reconstruct the account associated leaf node
22: **for** $i = 1, i < \log_2 n + 1, i \leftarrow i+1$ **do**
23: **if** $\overline{\texttt{ANodes}}[i].v < 0$ **then** **return** FALSE
24: **end if**
25: $\texttt{Node}'.v \leftarrow \texttt{Node}'.v + \overline{\texttt{ANodes}}[i].v$
26: **if** $\overline{\texttt{ANodes}}[i].t = 0$ **then** $\texttt{Node}'.h \leftarrow \mathsf{H}(\texttt{Node}'.v||\overline{\texttt{ANodes}}[i].h||\texttt{Node}'.h)$
27: **else** $\texttt{Node}'.h \leftarrow \mathsf{H}(\texttt{Node}'.v||\texttt{Node}'.h||\overline{\texttt{ANodes}}[i].h)$
28: **end if**
29: **end for**
30: **if** $\texttt{Node}' = \texttt{Nodes}[1]$ **then** **return** TRUE
31: **else** **return** FALSE
32: **end if**
33: **end procedure**

The exchange invokes the procedure FORMTREE to establish a binary tree with n leaves, a.k.a, the summation Merkle tree. The height of the tree is $\log_2 n+1$

and the tree includes $2n-1$ nodes in total. Every node consists of (v, h) where v is a balance value and h is a hash result under a collision-resistant hash function H. Every node holds a tag to identify its position, i.e., whether it is a left or right child of its parent on the tree. Using the $\mathsf{RandPermutate}()$ function, each leaf node is randomly targeted to a distinct account in $\mathcal{L}$. A leaf node contains an account balance v and also the hash h of v concatenating with the account login ID and the account nonce. Each internal node contains a balance equals to the sum of its child balances, and also a hash of its balance and its child hashes.

All the $2n-1$ nodes on the summation Merkle tree are filled up in a bottom-up order, and hence the balance of the root node is the summation of all leaf node balances, i.e. the summation of all client account balances in $\mathcal{L}$. The exchange publishes the root node for subsequent validation.

As a client, who has deposited in an exchange, can invoke the procedure VERIFYPROOF to check the validity of a proof of liabilities received from the exchange. The received proof, i.e. the output from the function $\mathsf{Receive}()$, is an authentication path of the leaf node associated with the client account. It is used to help the client to reconstruct a path, where the leaf node of the reconstructed path consists of the client requested account balance v and the hash h, obtained from the function $\mathsf{GetAccountInfo}()$. The client compares the computed root of the reconstructed path with the published root. If they are equal, this verification passes.

For a better understanding, we illustrate a summation Merkle tree with $n = 4$ in Fig.1, i.e., there are four accounts named A, B, C, D in exchange's list $\mathcal{L}$. The proof (i.e., the authentication path) for the left-most leaf node associated client account (i.e., account D) is written in gray background .

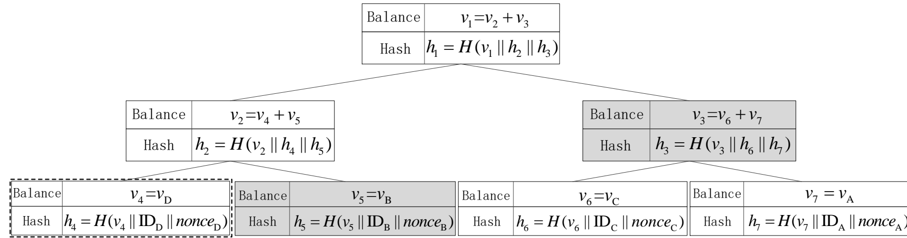

Fig. 1. A summation Merkle tree with 4 leaf nodes.

3 Attack and Improvements

We present an attack on the Maxwell protocol as well as the summation Merkle tree. Our attack results in exchanges to trick most of their clients that their account balances have been summed up into the published liabilities, whereas it only contains a small portion of the total amounts. We give out a lower bound of the amount of money that a valid proof of liabilities can reach using our attack on the Maxwell protocol, without the notice of any client in theory. Furthermore, we

propose two improvements to repair this problem and present a formal security proof for the improvement that is closest in spirit to the Maxwell protocol.

Definition 1. *For a decentralized proof of liabilities scheme that clients independently verify they are in the dataset, we call the scheme a secure proof of liabilities scheme if the claimed liabilities are no less than the sum of the amount of money in the dataset when no verification fails.*

Threat Model. We consider the threat model that an adversary (i.e., an exchange) holding $n = 2^s$ client accounts[1], where s is a positive integer, privately maintains and updates a list that stores all his client account information. Each entry of the list stores an account information and every account balance is at least zero. When the adversary making a proof of liabilities, he communicates with each client privately to agree on a fresh nonce and adds the nonce into the associated entry of his list. Each client receives a proof from the adversary separately. For the consideration of privacy, no client will share his account information or received proof with others.

3.1 Our Attack

Following the Maxwell protocol, a proof of liabilities received by a client consists of an authentication path on a summation Merkle tree. A proof is created by an exchange using the summation Merkle tree established by the exchange. Because the list used to establish the tree is preserved and updated by the exchange privately, the summation Merkle tree is completely private except for the root node that being published. Therefore, a client cannot verify the integrity of nodes on the received authentication path, and hence the integrity of those nodes cannot be guaranteed. An adversary can modify the contents of nodes on an authentication path before sending it to a client, especially change the node balances. To pass client verifications, the adversary changes the balances of internal nodes, including the root node balance, when establishing a summation Merkle tree. As long as each client's computed root equals to the root published by the exchange, the attack will not be discovered.

Our attack employs the above observation result and breaks the cumulative relation between an internal node and its children, which leads to the liabilities value published by the adversary is less than the actual one. This breaks the security goal for a secure proof of liabilities scheme as defined in Definition 1.

We first depict our attack in the case that no client owns two accounts in an exchange. we show the procesure done by the adversary in Algorithm 2. Clients follow the same verification procedure as in the Maxwell protocol. The idea behind our attack is that an adversary provides to its clients with authentication paths on which the node balances are *inconsistent* with, or more specifically less than, the values they actually contain. Meanwhile, the adversary remains the sum of

[1] When $n \neq 2^s$ for any positive integer s, an exchange can add some dummy accounts into $\mathcal{L}$ to resolve this. The balance of each dummy account could be a small positive number. This ensures that the added dummy accounts will not affect the published liabilities of the exchange.

sibling node balances and node hashes unchanged. Thereby, verifications can still pass without breaking the collision resistance of hash functions.

When establishing a tree, the adversary invokes the procedure $\overline{\text{FORMTREE}}$ and operates in a tricky way that instead of letting an internal node balance be the sum of its child balances, the adversary sets the balance v of each internal node to be a value no less than the maximum of its child balances[2], i.e., $max\{v_l, v_r\} \leq v \leq v_l + v_r$, as shown by the function ChooseValue() in Algorithm 2, and meanwhile keeps the node hash unchanged, i.e., $h = \mathsf{H}(v||h_l||h_r)$.

Algorithm 2 Our attack on the Maxwell protocol

Input: $n = 2^s$: number of client accounts; $\mathcal{L}$: all account information list; H: a collision resistant hash function

Output: A liabilities proof for a client

1: **procedure** $\overline{\text{FORMTREE}}$
2:     Nodes $\leftarrow$ empty ▷ Create an empty binary tree with $2n-1$ nodes
3:     $\tilde{\mathcal{L}} \leftarrow$ RandPermutate($\mathcal{L}$)
4:     **for** $i = n, i < 2n, i \leftarrow i+1$ **do** ▷ For each leaf node
5:         Nodes$[i].v \leftarrow \tilde{\mathcal{L}}[i].v$, Nodes$[i].h \leftarrow \mathsf{H}(\tilde{\mathcal{L}}[i].v||\tilde{\mathcal{L}}[i].\text{ID}||\tilde{\mathcal{L}}[i].nonce)$
6:     **end for**
7:     **for** $j = n-1, j > 0, j \leftarrow j-1$ **do** ▷ For each internal node
8:         $k \leftarrow$ ChooseValue($[max\{$Nodes$[2j].v$, Nodes$[2j+1].v\}$, Nodes$[2j].v+$Nodes$[2j+1].v]$) ▷ Choose a value from the input interval
9:         Nodes$[j].v \leftarrow k$
10:         Nodes$[j].h \leftarrow \mathsf{H}($Nodes$[j].v||$Nodes$[2j].h||$Nodes$[2j+1].h)$
11:     **end for**
12:     **for** $i = 1, i < 2n, i \leftarrow i+1$ **do**
13:         **if** i is odd **then** Nodes$[i].t = 1$ ▷ Identify a node's position that "1" indicates the node is a right child
14:         **else** Nodes$[i].t = 0$ ▷ "0" indicates the node is a left child
15:         **end if**
16:     **end for**
17:     Publish Nodes[1]
18: **end procedure**
19:
20: **procedure** CREATEPROOF ▷ Send the proof to a client
21:     Node$'$ $\leftarrow$ GetAccountInfo()
22:     $\overline{\text{ANodes}} \leftarrow$ GetAuthPath(Nodes, Node$'$)
23:     $\overline{\text{PNodes}} \leftarrow$ GetPath(Nodes, Node$'$)
24:     **for** $i = 1, i < \log_2 n + 1, i \leftarrow i+1$ **do**
25:         $\overline{\text{ANodes}}[i].v \leftarrow \overline{\text{PNodes}}[i+1].v - \overline{\text{PNodes}}[i].v$ ▷ Change the balances of nodes on the authentication path
26:     **end for**
27:     **return** $\overline{\text{ANodes}}$
28: **end procedure**

[2] The adversary has no motivation to choose v s.t. $v > v_l + v_r$ which will increase its apparent liabilities.

The adversary creates a liabilities proof for each client by firstly reconstructing the associated leaf node, as shown by the GetAccountInfo() function. The adversary extracts the authentication path and the path of the reconstructed leaf node from the established tree, by invoking the GetAuthPath() and GetPath() functions separately. Before sending the authentication path to a client, the adversary changes the balance of each node on the authentication path to be the difference between its parent and sibling node balances.

We go through a simple scenario with $n = 4$ that four accounts A, B, C, D are included as an example to our attack. The established tree from executing $\overline{\text{FORMTREE}}$ is shown in Fig.2.

Without loss of generality, when C, whose account is targeted to the left-most leaf node on the tree in Fig.2, doing a verification, C downloads the root node (v'_1, h'_1) published by the adversary, and receives a liabilities proof $\overline{\texttt{ANodes}} = \{(v^r_5 = v'_2 - v_4, h^r_5 = h_5), (v^r_3 = v'_1 - v'_2, h^r_3 = h'_3)\}$ which is the authentication path of C's leaf node. C reconstructs the path of his leaf node in a bottom-up order. C uses his account information to obtain his leaf node (v_4, h_4). By the use of $\overline{\texttt{ANodes}}[1] = (v^r_5, h^r_5)$, which is the first node on $\overline{\texttt{ANodes}}$, C computes and obtains $(\widetilde{v'_2}, \widetilde{h'_2})$ that:

$$\begin{cases} \widetilde{v'_2} = v_4 + v^r_5 = v'_2 \\ \widetilde{h'_2} = H(\widetilde{v'_2}||h_4||h^r_5) = h'_2. \end{cases}$$

Similarly, using $\overline{\texttt{ANodes}}[2] = (v^r_3, h^r_3)$, C computes and obtains $(\widetilde{v'_1}, \widetilde{h'_1})$ that:

$$\begin{cases} \widetilde{v'_1} = \widetilde{v'_2} + v^r_3 = v'_1 \\ \widetilde{h'_1} = H(\widetilde{v'_1}||\widetilde{h'_1}||h^r_3) = h'_1. \end{cases}$$

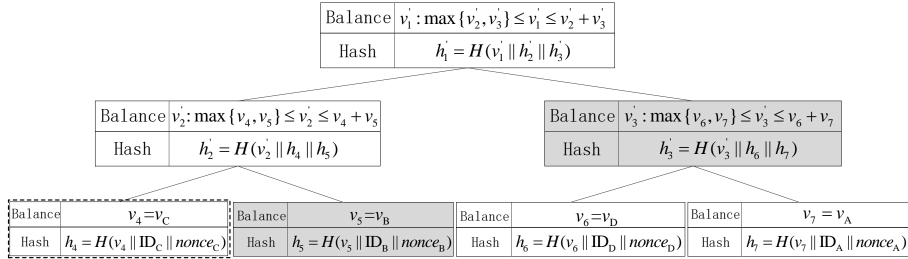

Fig. 2. A tree with 4 leaf nodes, resulting from running procedure $\overline{\text{FORMTREE}}$.

Due to $(\widetilde{v'_1}, \widetilde{h'_1}) = (v'_1, h'_1)$, the procedure VERIFYPROOF will return TRUE. It is easy to check that each client verification will return the same result. However, the published liabilities v'_1 only covers a part of the total amount of all balances in $\mathcal{L}$.

In the case that a client owns multiple accounts in an exchange, the client can reconstruct several paths during a proof of liabilities. These paths will certainly intersect at some internal nodes, which means that the client can obtain two pairs

of the child nodes for each intersection node. Although the verification procedure VERIFYPROOF will not check the consistency of different paths, the client may discover an attack if the exchange still provides inconsistent authentication paths.

Our attack is still feasible in this case. The results in [1,10,12,17] showed that it's possible to associate many cryptocurrency accounts with each other. Also, large centralized services such as exchanges and wallet services are capable of identifying and tracking user activity using techniques like cookies [23] and local storage [21]. Once the accounts are linked, the adversary can converge each group of linked accounts under a minimal subtree and keeps all the nodes on a subtree unchanged. The adversary modifies the remaining nodes on the summation Merkle tree and creates a proof for each client accordingly. As long as no client holds all accounts in $\mathcal{L}$, our attack is still feasible on the Maxwell protocol. However, if there is a client holding all accounts in an exchange, there is no need for the exchange to make a proof of liabilities.

Following the above analysis, we can easily come to Theorem 1 which gives a lower bound of the amount that a valid proof of liabilities can reach with our attack on the Maxwell protocol, without the notice of any client in theory.

Theorem 1 (Lower Bound of the Liabilities). *When using the Maxwell protocol to prove liabilities, the lower bound of the liabilities an exchange can claim, without the notice of any client, is the maximum value of all its account balances.*

Once an internal node balance value is less than the maximum value of its child balances, there must be an authentication path containing a node whose balance is a negative number. This makes the procedure VERIFYPROOF to return FALSE. We remark that, in reality, *it is wise for the adversary not to touch the lower bound*. Otherwise, it may lead to a notice of clients with large deposits, especially for the one with the largest deposit who will discover that the claimed liabilities equal to his account balance.

3.2 Improving the Maxwell Protocol

Ideally, a robust proof of liabilities would be designed to resist attacks like the aforementioned one and satisfy Definition 1. Since our attack yields such proof of liabilities are away from the original intention, the Maxwell protocol should be amended. We suggest two solutions to address this problem, if adopted, can resist our attack.

Solution 1. A natural solution is to add an additional proof when an exchange sending the proof of liabilities to its clients, in order to prove the integrity of nodes on an authentication path, i.e., to prove that these node hashes are commitments to their balances. Also, the verification process includes a step to check the validity of the additional proof.

This can be easily handled by sending the complete input data of the hash function of each node on an authentication path, but this can result in a privacy leakage. A smarter way is to adopt Zero-Knowledge Proof (ZKP) technology. Let $\mathtt{R}$ be a fixed NP-hard relation with the corresponding language $\mathtt{L} = \{h : \exists x,\ s.t.\ (x, h) \in \mathtt{R}\}$. We denote the zero-knowledge proof of the preimage of the

hash function H by $\pi = \texttt{ZK.Prove}\{(x) : h = \mathsf{H}(v||x)\}$. Receiving a proof π, a client can invoke the verification algorithm of ZKP to check its validity. However, the introduction of ZKP would lose the simplicity of the original Maxwell protocol. Readers who are interested in the realization of this zero-knowledge proof can refer to [9] for details.

Solution 2. A better solution which is closest in spirit to the original Maxwell protocol is to modify the input format of the hash function for *internal nodes* including both its child balances unsummed, i.e., $h = \mathsf{H}(v_l||v_r||h_l||h_r)$. Correspondingly, the input format of the hash function when doing a verification should be modified to include an internal node both child balances unsummed too.

We present Theorem 2 with a formal proof to state that this solution can help the Maxwell protocol to achieve the security requirement defined in Definition 1.We first give and proof a lemma. In the following proofs, we let $\boldsymbol{n}^k = (v^k, h^k)$ indicate the node at position k that on the established tree, and $\boldsymbol{n}_i^k = (v_i^k, h_i^k)$ indicate the node at position k that computed or received by a client C_i.

Lemma 1. *If the Maxwell protocol adopting the modification in Solution 2, every intersection node of two successfully verified paths remains the same as the one on the established summation Merkle tree, assuming the collision resistance of the hash functions.*

Proof. Suppose there exists at least one intersection node not satisfying this lemma. Let t be an unique identifier for the position of such an intersection node with the *maximum* height, then there exists two paths associates with client C_i and client C_j respectively, s.t., $\boldsymbol{n}_i^t \neq \boldsymbol{n}_j^t$ or $\boldsymbol{n}_i^t = \boldsymbol{n}_j^t \neq \boldsymbol{n}^t$, where $\boldsymbol{n}_i^t$ and $\boldsymbol{n}_j^t$ are the nodes at position t computed by C_i and C_j respectively.

Let p (resp. s) be an unique identifier for the position of the parent (resp. sibling) node of the node at position t. According to the definition of the node at position t, the node at position p is an intersection node of C_i and C_j's paths such that $\boldsymbol{n}_i^p = \boldsymbol{n}_j^p = \boldsymbol{n}^p$, where $\boldsymbol{n}_i^p$ and $\boldsymbol{n}_j^p$ are the nodes at position p computed by C_i and C_j respectively. Let $\boldsymbol{n}_i^s$ and $\boldsymbol{n}_j^s$ be the nodes at position s on the received authentication paths of C_i and C_j respectively. Because $\boldsymbol{n}_i^p = \boldsymbol{n}_j^p = \boldsymbol{n}^p$, we have $h_i^p = h_j^p = h^p$, thus $\mathsf{H}(v_i^t||v_i^s||h_i^t||h_i^s) = \mathsf{H}(v_j^t||v_j^s||h_j^t||h_j^s) = \mathsf{H}(v^t||v^s||h^t||h^s)$.

Due to $\boldsymbol{n}_i^t \neq \boldsymbol{n}_j^t$ or $\boldsymbol{n}_i^t = \boldsymbol{n}_j^t \neq \boldsymbol{n}^t$, we have $(v_i^t, h_i^t) \neq (v_j^t, h_j^t)$ or $(v_j^t, h_j^t) \neq (v^t, h^t)$. Let $x_1 = v_i^t||v_i^s||h_i^t||h_i^s, x_2 = v_j^t||v_j^s||h_j^t||h_j^s$ and $x_3 = v^t||v^s||h^t||h^s$. We have $x_1 \neq x_2$ or $x_2 \neq x_3$, but $\mathsf{H}(x_1) = \mathsf{H}(x_2) = \mathsf{H}(x_3)$. It conflicts with the assumption of the collision resistant of the hash function H. Hence, we proved this lemma. □

Theorem 2. *The Maxwell protocol is a secure proof of liabilities scheme when adopting the modification in Solution 2, assuming the collision resistance of hash functions.*

Proof. Suppose there exists at least one node on the established summation Merkle tree of the modified Maxwell protocol whose balance is less than the sum of its child balances, and no verification of the proof of liabilities fails. Let t be an unique identifier for the position of such a node with the *minimum* height. Because an adversary has no motivation to create an internal node whose balance is larger than its child balances. If this node do exist, when the balances are summed up

in a bottom-up order to the root node, the root balance can be less than the sum of all the leaf balances. Therefore, the claimed liabilities are less than the sum of all account balances in the adversary's list. This makes the modified Maxwell protocol not a secure proof of liabilities scheme.

Let $\boldsymbol{n}^t = (v^t, h^t)$ be the node at position t on the established tree. Let further l (resp. r) be an unique identifier for the position of $\boldsymbol{n}^t$'s left (resp. right) child node. Thus, we have $v^t < v^l + v^r$, where v^l (resp. v^r) is the balance of node at position l (resp. r) on the tree. For client C_i (resp. C_j) whose path goes through the node at position l (resp. r), he reconstructs this node $\boldsymbol{n}_i^l$ (resp. $\boldsymbol{n}_j^r$) and receives a node $\hat{\boldsymbol{n}}_i^r$ (resp. $\hat{\boldsymbol{n}}_j^l$) at position r (resp. l) on his authentication path.

When the node at position l is a leaf node, C_i reconstructs this node using his account information. Thus, $\boldsymbol{n}_i^l = \boldsymbol{n}^l$. When the nodes at position l is an internal node, it must be an intersection node of two paths. According to Lemma 1, $\boldsymbol{n}_i^l = \boldsymbol{n}^l$ is also satisfied. Hence, we have $v_i^l = v^l$. Similarly, we have $\boldsymbol{n}_j^r = \boldsymbol{n}^r$ and $v_j^r = v^r$.

Because the node at position t is an intersection node of C_i and C_j's paths, according to Lemma 1, we have $\boldsymbol{n}_i^t = \boldsymbol{n}_j^t = \boldsymbol{n}^t$, where $\boldsymbol{n}_i^t$ and $\boldsymbol{n}_j^t$ are the nodes at position t computed by C_i and C_j respectively.

When C_i reconstructs $\boldsymbol{n}_i^t$, he computes $v_i^t = v_i^l + \hat{v}_i^r$, and $h_i^t = \mathsf{H}(v_i^l||\hat{v}_i^r||h_i^l||\hat{h}_i^r)$. Because $v_i^t = v^t < v^l + v^r$ and $v_i^l = v^l$, we have $\hat{v}_i^r \neq v^r$. Similarly, we have $h_j^t = \mathsf{H}(\hat{v}_j^l||v_j^r||\hat{h}_j^l||h_j^r)$, $v_j^r = v^r$ and $\hat{v}_j^l \neq v^l$ when C_j reconstructs $\boldsymbol{n}_j^t$. Therefore, we can construct $x_1 = v_i^l||\hat{v}_i^r||h_i^l||\hat{h}_i^r$ and $x_2 = \hat{v}_j^l||v_j^r||\hat{h}_j^l||h_j^r$ with $x_1 \neq x_2$, but $\mathsf{H}(x_1) = \mathsf{H}(x_2)$. This conflicts with the assumption of the collision resistant of the hash function H. Hence, we proved Theorem 2. □

4 Attack Instance One: The Provisions Scheme

The Provisions [6] is a privacy-preserving proofs of solvency scheme, which consists of a privacy-preserving proofs of liabilities scheme and a privacy-preserving proofs of reserves scheme. Compared to the Maxwell protocol, the Provisions proof of liabilities enable to convince clients that their balances have been summed up in the liabilities of the exchange without revealing the total asset of this exchange or any other account balance.

Each time the exchange making a proof of liabilities, it publishes a list that each entry consists of an account information $\langle \text{CID}, z, \pi \rangle$. $\text{CID} = \mathsf{H}(\text{ID}||n)$ is a hash result of the account login ID and a nonce n, where n is chosen by the exchange. $z = g^v h^r$ is a Pedersen commitment to the account balance v, where r represents the string that can be used to open/reveal the commitment and is chosen by the exchange. π is a knowledge proof showing that z is well-formed. Due to the homomorphism of the Pedersen commitment scheme, the commitment to the total liabilities of the exchange is the product of all entries' commitments.

When a client verifies a liabilities proof, he downloads the list. He checks the entry associates with his account, using his account information and the received nonce n and string r from the exchange. He also checks the validity of other entries by verifying each entry's knowledge proof. Finally, the client compares the product of all entries' commitments with the one published by the exchange, and returns TRUE if they are equal.

To reduce the computation overhead of per-verification from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ (where n is the size of the list), the authors presented an extension to the Provisions proof of liabilites in their full paper [5]. They rearrange the account information into the summation Merkle tree of the Maxwell protocol. Each leaf node contains (CID_i, z_i) for some client i, and each internal node contains a commitment to the sum of its child balances together with a hash of its children.

Similar to the Maxwell protocol that instead of publishing a list, an exchange publishes the root node of the tree. A verification in this extension only incurs an $\mathcal{O}(\log n)$ computation overhead instead of checking all n entries of a list.

Our attack still holds on this modified version of the summation Merkle tree used in the Provisions scheme. An adversary $\mathcal{A}$ (i.e., an exchange) establishes a binary tree, similar to Fig.3. Each internal node stores a commitment to the *maximum* of its child balances, and also a hash of its children. All leaf nodes remains unchanged. Although an internal node commitment does not equal to the product of its child commitments, $\mathcal{A}$ can establish such a tree because the r of each commitment z is chosen by $\mathcal{A}$ and $\mathcal{A}$ knows all its client account balances.

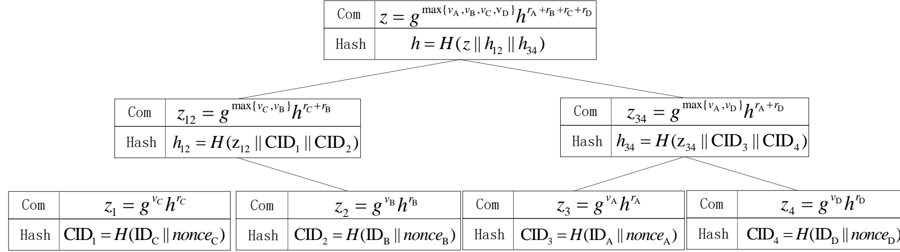

Fig. 3. An established tree when $n = 4$, resulting from an adversary conducting an attack on the extension of the Provisions proof of liabilities.

When creating a proof for some client i, $\mathcal{A}$ finds out the leaf node that associates with i's account and extracts its authentication path. Before sending a proof to client i, $\mathcal{A}$ modifies the commitments of nodes on this authentication path and remains other parts unchanged. A modified commitment is a commitment to the difference between its parent node committed value and its sibling node committed value. It is easy to check that when client i follows the procedure VERIFYPROOF to verify a received proof, it will return TRUE. Therefore, the extension of the Provisions proof of liabilities is not a secure proofs of liabilities scheme.

5 Attack Instance Two: The DAM Scheme

5.1 Scheme Overview

The DAM scheme [3] is a lottery-based micropayment scheme for the decentralized cryptocurrency – Zerocash [18]. Considering "micro" sums of money of the payment, micropayments have relatively high transaction costs to incentivize blockchain miners to behave honestly. With this in mind, the DAM scheme uses

the idea of probabilistic payments to deal with this problem that instead of sending a transaction of value v, the user sends a lottery ticket whose *expected payout* is v. When the lottery ticket's winning probability is p, for $1/p$ transactions on average, there is only one transaction with value v/p will result in a money transfer recorded on the blockchain.

In the DAM scheme, every time a user is making a transaction with a merchant, the user sends a lottery ticket to the merchant. Considering a malicious user, he may issue the same lottery ticket to several merchants or even to himself, that multiple of them might win but only one will be able to cash in this ticket, known as double/multiple spending attack. In most micropayment applications, like streaming services, they often require fast responses, and these in turn result in merchants checking the validity of payments *offline*. Because not all double/multiple spending can be detected timely, double/multiple spending cannot be prevented in offline payments.

Given this, the DAM scheme uses the "detect-and-punish" approach to disincentivize double spending. If a user double spends, his behavior can be detected and he will lose some money as a punishment, for example, his deposit will be frozen. Therefore, every lottery ticket should be bound to a *single deposit* of the user. When making a transaction using this ticket, the user also sends a unique 2-out-of-n secret share of the deposit. In case of the ticket is used twice, the secret of the deposit can be recovered and the deposit with this unique secret will be frozen, which renders a punishment on the user.

The "detect-and-punish" approach is effective only if the disadvantage of being punished outweigh the advantage of double spending. Hence, the size of a deposit should be larger than the additional utility gained by double spending. The DAM scheme associates with each deposit a "receiver address set $\mathcal{R}$", which consists of the information of merchants that the deposit is allowed to engage in payments with. Each element of $\mathcal{R}$ contains $(apk, w, \boldsymbol{d})$, where apk is a merchant Zerocash address, w is the cumulative value of transactions that can be accepted by the merchant within a time period (e.g., a day), and $\boldsymbol{d}$ represents other auxiliary data of the merchant. They are publicly available. Because the cumulative maximum value of transactions that can be accepted by merchants in $\mathcal{R}$ using the deposit, within a time period, is $\sum_{i=1}^{n} w_i$, if there are n merchants included in $\mathcal{R}$. To make this approach effective, the value of a deposit should be at least $\sum_{i=1}^{n} w_i$.

The DAM scheme deals with this problem by adopting the Maxwell protocol to establish a summation Merkle tree, as described in their full paper [2]. On the established tree, each leaf node consists of a element of $\mathcal{R}$, and each internal node stores the sum of its child values together with a hash over its children.

5.2 Multi-Spending Attack

By establishing a summation Merkle tree, the DAM scheme is going to achieve two purposes: (1) to provide an efficient set membership proof, i.e., the user will only be able to transact with those merchants for whom he can produce a valid authentication path, and (2) to provide a deposit validity proof, i.e., the value of the deposit is enough to disincentivize double spending.

The full paper [2] of the DAM scheme also provides a proof via induction to show that the root of a summation Merkle tree contains a value at least as large

as $\sum_{i=1}^{n} w_i$, i.e., the Maxwell protocol is secure. Due to this, the authors believed that (2) can be achieved. We put the proof from [2] below:

"We proceed via induction on n. The case for $n = 1$ is trivial. Assume that the hypothesis is true for $n = k$. Then we prove that if the $k + 1^{th}$ path is valid, then the sum at the root is $\sum_{i=1}^{k+1} w_i$.

Let the $k + 1^{th}$ path intersect a previous path p_i at an internal vertex v, and let the values of the two paths at v be different. Since both paths are valid, the hashes at v must be equal. But these two facts cannot be reconciled unless the collision resistance of the hash function is broken. Since we assume collision resistance, this means that the values of v in both p_i and p_{k+1} must be equal. This means that $v \geq w_i + w_{k+1}$. Propagating this sum up the tree, we obtain that the sum at the root must be at least as large as $\sum_{i=1}^{k+1} w_i$"

However, our attack in Section 3.1 is a counterexample to the above claimed result, and hence the proof is defective. The result in the above proof that the intersection node of any two paths must be equal is correct, which Lemma 1 of this paper comes to the same result. The problem is, this result cannot deduce that the value of the intersection node is no less than the sum of its children.

We mention that the summation Merkle tree established in the DAM scheme is slightly different from the one in the Maxwell protocol as described in Algorithm 1. In the DAM scheme, a leaf node contains a full content of an element of $\mathcal{R}$, i.e. $(apk_i, w_i, \boldsymbol{d}_i)$ for some i, rather than only containing w_i and a hash result over $(apk_i, w_i, \boldsymbol{d}_i)$. Our attack still holds under this difference with a few minor changes: When an adversary $\mathcal{A}$, as a user, mounting an attack, he sets each leaf node consists of an element of $\mathcal{R}$ and the value of each internal node of height two equals to the sum of its child values. For the rest of internal nodes, $\mathcal{A}$ follows the instructions in FORMTREE. Therefore, the root value equals to the maximum sum of two sibling leaf node values. When $\mathcal{A}$ creates a proof for some merchant i, he follows the procedure CREATEPROOF in Algorithm 2 to obtain a modified authentication path of the leaf node that associates with i's information, but remains the content of the leaf node on the authentication path unchanged. It is easy to check that when merchant i invokes the procedure VERIFYPROOF to verify a received proof, the procedure will return TRUE.

The reason for the changes comes from the fact that every merchant information $(apk, w, \boldsymbol{d})$, is available to everyone, and hence a merchant can verify the integrity of a received leaf node. This renders that a valid proof cannot include a modified leaf node, thus all parent nodes of leaf nodes, i.e., the internal nodes of height two, should remain unchanged.

An adversary establishes a modified summation Merkle tree described above, whose root value equals to the maximum sum of two sibling leaf node values. He makes transactions to several merchants simultaneously, and multi-spends the same lottery ticket to these merchants. Because all of the merchants having received proofs of liabilities can pass the verifications, they will believe in the validity of the bound deposit and provide services to the adversary. However, multiple merchants may win the lottery ticket and only one will be able to cash in the lottery ticket. Of course, the adversary will lose the deposit, he can also gain additional utility by mounting a multi-spending attack.

Next, let's calculate the additional utility an adversary $\mathcal{A}$ can obtain from a multi-spending attack. Only if this value is larger than zero, $\mathcal{A}$ has the motivation to mount such an attack. We remark that for an honest user, he only needs to pay for the d winners when conducting k micropayments ($k \geq d$), rather than all k micropayments because a lottery ticket is essentially probabilistic. Therefore, in our multi-spending attack, for the merchant r who wins the lottery ticket and gets paid, as well as other $k - d$ merchants who do not win, $\mathcal{A}$ is honest. For the remaining $d - 1$ merchants who won the lottery but didn't get paid, $\mathcal{A}$ is evil. Thus, $\mathcal{A}$'s additional utility comes from those $d - 1$ merchants.

Let $\mathcal{I}$ be a set containing all d winners' indexes, r be the index of the merchant who actually cash in the ticket, and V be the value of $\mathcal{A}$'s deposit. Then $\mathcal{A}$'s additional utility (AU) is,

$$\mathrm{AU} = \sum_{i \in \mathcal{I} \setminus \{r\}} w_i - V \tag{1}$$

Next, we analyze the condition that $\mathcal{A}$ can benefit from multi-spending. For simplicity, we assume a flat model where $\forall (i \neq j) \wedge (w_i, w_j \in \mathcal{R}), w_i = w_j = w$. We have,

$$V = max\{w_i + w_j : \forall w_i, w_j \in \mathcal{R} \wedge i \neq j\} = 2w \tag{2}$$

$$\sum_{i \in \mathcal{I} \setminus \{r\}} w_i = (d-1)w \tag{3}$$

According to Eq. (1)-(3), when $d - 1 > 2$, i.e., $d > 3$, AU > 0. In this condition, $\mathcal{A}$ has the motivation to mount a multi-spending attack.

Now, we are interested in the probability that more than λ (=3) merchants can win the lottery ticket. Suppose the probability that each merchant can win is p, and there are k merchants in total. To calculate the probability of a successful attack, we use the cumulative binomial distribution [22] where X is the random variable that represents the number of the winning merchants:

$$P[X > \lambda] = 1 - P[X \leq \lambda] = 1 - \sum_{j=0}^{\lambda} \binom{k}{j} p^j (1-p)^{k-j} \tag{4}$$

Table 1. Expected probability of a successful attack.

$p \mid k$	20	30	40	50
0.1	0.133	0.353	0.577	0.750
0.2	0.589	0.877	0.972	0.994

Table 1 displays the evaluation results of Eq. (4) for various window sizes k both in $p = 0.1$[3] and $p = 0.2$. From a security perspective, the result in Table 1 suggests that the multi-spending attack is a big threat to the DAM scheme.

[3] The value p is not specified in the DAM scheme. Here, we adopt p according to another micropayment scheme [16] using the same idea of probabilistic payments.

6 Conclusion

Proofs of liabilities are used for exchanges holding a large amount of client deposits to prove the sums of money they should own. Our results show that the Merkle approach to prove liabilities is vulnerable. We formalized the Maxwell protocol and its data structure – the summation Merkle tree. We presented an attack on the Maxwell protocol that enables an exchange controlling a total of n client accounts can provide valid liabilities proofs including only one account balance. We further gave a lower bound of the amount that a verifiable proof of liabilities can reach without the notice of any client in theory. To repair this problem, we suggested two improvements and presented a security proof to the improvement that is closest in spirit to the Maxwell protocol. Furthermore, we showed that our attack can be carried over to two use cases of the Maxwell protocol and the underlying summation Merkle tree: the DAM scheme and the Provisions scheme. For both applications, we suggest to use one of our improvements to fix the attacks.

References

1. E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating User Privacy in Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 34–51. Springer, 2013.
2. A. Chiesa, M. Green, J. Liu, P. Miao, I. Miers, and P. Mishra. Decentralized Anonymous Micropayments. Cryptology ePrint Archive, Report 2016/1033, 2016. http://eprint.iacr.org/.
3. A. Chiesa, M. Green, J. Liu, P. Miao, I. Miers, and P. Mishra. Decentralized Anonymous Micropayments. In *Advances in Cryptology – EUROCRYPT 2017*, pages 609–642. Springer, 2017.
4. CoinDesk. OKCoin Reveals BTC Reserves of 104% as China's Exchanges Undergo Audits. https://www.coindesk.com/okcoin-reveals-btc-reserves-104-chinas-exchanges-undergo-audits/.
5. G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh. Provisions: Privacy-preserving Proofs of Solvency for Bitcoin exchanges. Cryptology ePrint Archive, Report 2015/1008, 2015. http://eprint.iacr.org/.
6. G. G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh. Provisions: Privacy-preserving Proofs of Solvency for Bitcoin Exchanges. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 720–731. ACM, 2015.
7. Dashcoin. Anonymous peer-to-peer Internet currency. http://dashcoin.info/.
8. C. Decker and R. Wattenhofer. Bitcoin Transaction Malleability and MtGox. In *Computer Security - ESORICS 2014*, pages 313–326. Springer, 2014.
9. M. Jawurek, F. Kerschbaum, and C. Orlandi. Zero-Knowledge Using Garbled Circuits or How To Prove Non-Algebraic Statements Efficiently. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 955–966. ACM, 2013.
10. P. Koshy, D. Koshy, and P. McDaniel. An Analysis of Anonymity in Bitcoin Using P2P Network Traffic. In *International Conference on Financial Cryptography and Data Security*, pages 469–485. Springer, 2014.
11. Kraken. Audit: Learn about Kraken's audit process. https://www.kraken.com/security/audit#verify.

12. S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A Fistful of Bitcoins: Characterizing Payments Among Men with No Names. In *Proceedings of the 2013 conference on Internet Measurement Conference*, pages 127–140. ACM, 2013.
13. I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.
14. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System (2008). https://bitcoin.org/bitcoin.pdf.
15. S. Noether. Ring Signature Confidential Transactions for Monero. Cryptology ePrint Archive, Report 2015/1098, 2015. http://eprint.iacr.org/.
16. R. Pass and a. shelat. Micropayments for Decentralized Currencies. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 207–218. ACM, 2015.
17. F. Reid and M. Harrigan. An analysis of Anonymity in the Bitcoin System. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom)*, pages 1318–1326. IEEE, 2011.
18. E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.
19. Uphold. Security at Uphold. https://support.uphold.com/hc/en-us/articles/203399367-Security-at-Uphold.
20. Vaultoro. Transparency. https://audit.vaultoro.com/.
21. M. web docs. window.localStorage. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.
22. Wikipedia. Binomial distribution. https://en.wikipedia.org/wiki/Binomial_distribution#Cumulative_distribution_function.
23. WIKIPEDIA. HTTP cookie. https://en.wikipedia.org/wiki/HTTP_cookie#cite_note-1.
24. Z. Wilcox. Proving Your Bitcoin Reserves. https://bitcointalk.org/index.php?topic=595180.0.