Security Mechanisms for Protection Against Double-Spending

DOUBLE-SPENDING ATTACKS are when an owner of an asset is able to transfer it to two different parties, often simultaneously. There are two broad categories of double-spending attacks:

- EXPIRED OWNERSHIP – *In this scenario, a party attempts to send an asset to another party after having already previously transferred its ownership away.*

- SIMULTANEOUS TRANSFERS – *A party may attempt to transfer the ownership of an asset to two distinct parties at the same time*

Generally, attacks of this nature work in cryptosystems where the recipient is unaware that the file had either already been transferred, or was in the process of being transferred to another. In particular, these attacks can only be successful when the sender is able to *hide information* from a recipient.

What information is being hidden? In the case of expired ownership, the transaction history of an asset must have omissions for the attack to be successful. For simultaneous transfers, knowledge of another transaction affecting a given asset is omitted.

Protection against expired ownership

TODA is designed to enforce the integrity of asset ownership by making double-spending impossible. TODA requires that a sender provide a fully-saturated *merkle proof of provenance* of any asset it has transferred to a recipient. This proof is a chain of data, aligned with the cycles present in the TODA chain, accounting for every opportunity where the file may have changed ownership. Each component of this merkle proof of provenance proves either that the asset was *transacted* to a new owner, or *not transacted*[1].

A *merkle proof of provenance (MPOP)* is titled after its role in establishing an asset's *provenance*, positively answering the question, *how did this file come to be mine?*. An MPOP which is able to account for the activity of a file during every cycle of its existence is termed *fully-saturated*.

[1] Structures which prove that an asset was not transacted are termed *null proofs*.

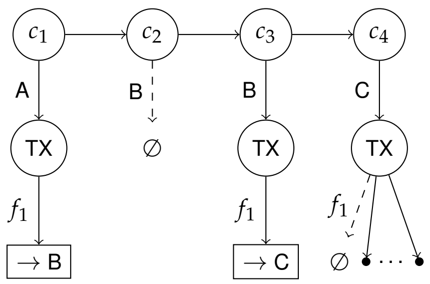

Figure 1: A *fully-saturated* merkle proof of provenance for $c_1 \ldots c_4$.

In figure 1, f_1 is transferred from A to B in c_1. In c_2, B provides a proof that they did not transact. In c_3, f_1 is transferred from B to C. In c_4, C provides a proof that they transacted, but that this transaction did not include f_1, thereby maintaining their ownership of the file.

In an attack scenario, B from figure 1 may attempt to transfer their ownership of f_1 back to A in the following cycle, c_5, after they already transferred it to C in c_3. In this case, A would examine the fully-saturated proof from c_1 to c_4 to readily discover that B is no longer the valid owner, and are not able to assign ownership of f_1.

Consider a similar case, but where B hides the fact that they transferred f_1 away in c_3 (figure 2). When A examines this proof, they would discover it was not fully-saturated, and failed to account for either the activity of B during c_3, or more specifically, any action pertaining to f_1 during that cycle. An analogous attack where information is omitted, but from the details of the transaction, rather than whether a transaction occurred, is shown in figure 3. This again fails, as the hidden information prevents B from demonstrating an uninterrupted proof of ownership.

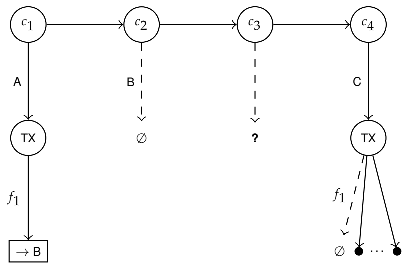

Figure 2: B attempts to hide their activity in c_3, but is unable to provide a fully-saturated proof of f_1.

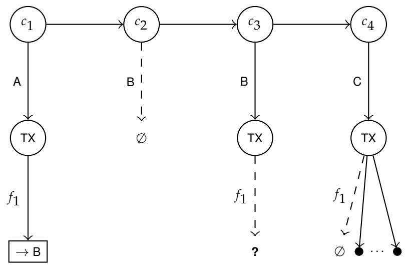

Figure 3: B attempts to hide the specifics of their transaction in c_3, but is unable to provide a valid null-proof of f_1 for this cycle.

Protection against simultaneous transfer

TODA's protection against simultaneous transfer stems directly from the fundamental architecture of the data representing a proof. In this way, TODA distinguishes itself from the concerns inherent in several cryptographic asset management schemes: **it is not possible to represent a simultaneous transfer attack in TODA**.

A valid proof concerning an asset during a given cycle requires two major components: a proof of the activity of its *owner* during that cycle, referred to as the *address proof*, and a proof of whether the file was transacted by the owner during that cycle, termed the *file proof*.

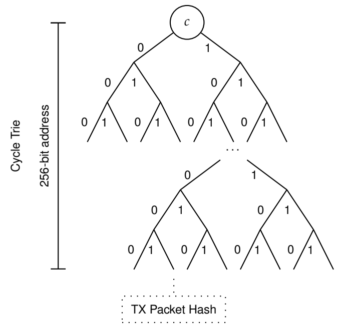

Figure 4: The *address proof* is a trimmed cycle trie, proving the value of a *transaction packet hash* for a given address in a given cycle.

Both the address proof and the file proof are derived from Merkle tries[2]. The address proof is a minimal subgraph[3] of a merkle trie which represents all owners who are transacting in a given cycle. The file proof is a minimal subgraph a merkle trie representing a map of all transacted assets to their destinations.

[2] A *Merkle trie* is a cryptographic data structure with renewed research and industrial interest following the repopularization of *Merkle trees* in Bitcoin and similar systems. A *Merkle trie* combines the properties of a *Merkle tree* with those of a *trie*, meaning that it both provides cryptographic integrity, as well as deterministic paths of its contents.

[3] A *minimal subgraph* of a merkle trie is the path from the root of the trie to the leaf of interest, and including the merkle root of each branching subtree. For further background on the properties of merkle tries and their use in TODA, refer to the *TODA Primer*.

In order to effect the simultaneous transfer of f_1 to both A and B, an attacker might aim to construct two valid file proofs for f_1. A file proof is extracted from the transaction trie, representing all files transacted by a given owner during a cycle. Table 1 shows the potential contents of a file proof where three files have been transacted. This table is converted into a trie where the *paths* of the trie represent the *asset hash*, and the values represent the *destination*.

Asset	Asset Hash	Destination
f_1	`0xbf01...`	**B**
f_2	`0x3493...`	**B**
f_3	`0xf930...`	**C**

Table 1: The data represented in **A**'s *transaction trie* in Cycle 1 (c_1) of the above example.

It is at this point that the attacker encounters a problem. Due to the deterministic path property of tries, they are unable to simply add (or hide) another value for f_1 within their transaction trie. In doing so, they would simply displace the initial value they had stored for f_1 in the structure.

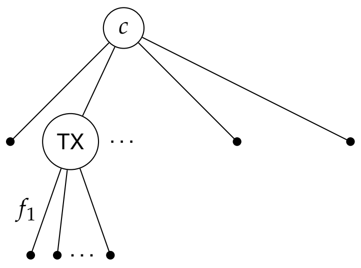

Figure 5: There is only a single slot in a trie data structure where a value can be stored for f_1 for a given address.

To arrive with a valid proof for each destination of f_1 then, the attacker would need to construct two distinct transaction tries: T_1, T_2. Unfortunately, this too, doesn't open an opportunity for the attacker to gain an advantage. For a transaction trie to have any validity, it must be attached to the cycle trie for a given cycle. And, as the cycle trie holds the same deterministic path properties as the transaction trie, it is not possible to simultaneously express a mapping to two distinct transaction hashes during a single cycle.

Even if the attacker attempts to have both $H(T_1)$ and $H(T_2)$ incorporated into a cycle trie simultaneously, we have seen that this is not a representation which is even supported at the level of the *data structure*. Furthermore, at the level of the *TODA Protocol*, it is an error for any participant to sign two distinct transaction tries within a given cycle, and those parties constructing the cycle trie will, in fact, discard *all* updates originating from that client.

Technical points of contact:

Adam Gravitis
Chief Technology Officer
adam.gravitis@todaqfinance.com

Dann Toliver
Co-Founder & Protocol Author
dann.toliver@todaqfinance.com