How to keep data private

This document describes some strategies for keeping data private in Midnight contracts. This is not an exhaustive list, but it should help you get started.

The most crucial thing to bear in mind is that, except for `[Historic]MerkleTree` data types, anything that is passed as an argument to a `ledger` operation in Compact, as well as all reads and writes of the ledger itself, are publicly visible and should be treated as such. What is public is the argument or ledger value itself, not the code that manipulates it. For instance:

```
export ledger items: Set<Field>;
export ledger others: MerkleTree<10, Field>;

// Reveals `item1`
items.insert(item1);
// Reveals the *value* of `f(x)`, but not `x` directly
items.member(f(x));
// The exception: Does *not* reveal `item2`, though someone that
// guesses the value of `item2` can check it!
others.insert(item2);
```


However, sometimes you need to reference shielded data in the public state. In those cases, one of the patterns below may hel

Hashes and commitments

The most basic approach to storing data in public, while keeping it shielded, is to store only a hash or commitment of data, rather than the full data itself.

Compact's standard library provides two primary primitives for this:

- `persistentHash`, a building block to hash binary data
- `persistentCommit`, a primitive for creating commitments from any Compact type.

Both of these effectively create a hash of their inputs, with `persistentHash` being limited to the `Bytes<32>` data type and `persistentCommit` hashing arbitrary data together with a `Bytes<32>` random value. Hashes guarantee that the input cannot be computed from the output, nor any information about the input guessed, unless the whole input is guessed. This is one reason the additional randomness input of `persistentCommit` is important: it prevents someone from guessing the value itself and checking that the hash matches. This is especially useful when there are a small number of possible values, such as an individual vote in an election.

The other advantage of randomness is that it prevents correlating equal values: even if I can't guess someone's password, for instance, I might recognize if the same hashed version appears twice, which might unintentionally leak information about who made a state change.

With sufficient randomness used, the commitment of a value can be stored on the ledger without revealing it.

Randomness and rounds in commitments

Fresh randomness for each commitment is desirable, but in some cases, it is possible to re-use existing randomness by guaranteeing that the *data* will never be the same for the same randomness. We use this in some of our example applications, where we reuse a secret key as a randomness source, together with a round counter to ensure unlinkability between rounds.

> CAUTION

> Be careful working with randomness! It's easy to get wrong, and erring on the safe side is generally advisable.

Authenticating with hashes

One of the most useful features of zero-knowledge proofs is that it's possible to emulate signatures just by using hashes in a circuit. That is, just by hashing a secret key and comparing that with a known 'public key', a contract can guarantee that only someone that knows the secret key can continue the transaction. For instance, here's a contract that allows only the creator to use it:

```
import CompactStandardLibrary;

witness secretKey(): Bytes<32>;

export ledger organizer: Bytes<32>;
export ledger restrictedCounter: Counter;
constructor() {
  organizer = publicKey(secretKey());
}

export circuit increment(): [] {
  assert(organizer == publicKey(secretKey()), "not authorized");
  restrictedCounter.increment(1);
}

circuit publicKey(sk: Bytes<32>): Bytes<32> {
  return persistentHash<Vector<2, Bytes<32>>>([pad(32, "some-domain-seperator"), sk]);
}
```

Making use of Merkle trees

Feedback

Merkle trees, exposed in Compact as the `MerkleTree<n, T>` and `HistoricMerkleTree<n, T>` types, are a very useful tool for shielding the values contained in a set. Their key feature is making it possible to assert publicly that some value is contained within the `MerkleTree`, without revealing *which* value this is.

This goes above and beyond having, for instance, a `Set<Bytes<32>>` storing commitments and testing if a commitment is inside it, because a `MerkleTree` does not reveal which entry's membership is proven. This property can be used, for instance, to authorize a set of secret keys to do specific operations, without each operation revealing which key was used to authorize it.

In practice, this works by a circuit proving that it has knowledge of a path to an inserted value in the tree and checking that the hash of this path matches the expected path of the tree.

The Compact standard library and compact JavaScript target ADTs provide tools for these operations. Specifically, examine the `MerkleTreePath<n, T>` type, the `merkleTreePathRoot<n, T>()` circuit, and the `pathForLeaf()` and `findPathForLeaf()` functions exposed on the `MerkleTree`/`HistoricMerkleTree` JavaScript state objects, as described in the ledger data types specification.

Together, they can be used as follows:

```
import CompactStandardLibrary;

export ledger items: MerkleTree<10, Field>;

witness findItem(item: Field): MerkleTreePath<10, Field>;

export circuit insert(item: Field): [] {
  items.insert(item);
}

export circuit check(item: Field): [] {
  const path = findItem(item);
```

```
  assert(items.checkRoot(merkleTreePathRoot<10, Field>(path.value)), "path must be valid");
}
```

With the `findItem` implementation:

```
function findItem(context: WitnessContext, item: bigint): MerkleTreePath<bigint> {
    return context.ledger.items.findPathForLeaf(item)!;
}
```

Note that `pathForLeaf` is preferable when possible, as it does not require an *O(n)* scan of the tree, although it does require knowledge of where the item was originally placed.

The distinction between `MerkleTree<n, T>` and `HistoricMerkleTree<n, T>` is that `checkRoot` for the latter accepts proofs made against prior versions of the Merkle tree. This is helpful if a tree has frequent insertions, as these otherwise invalidate old proofs, although `HistoricMerkleTree` is not suitable if items are frequently removed or replaced, as this could lead to proofs being considered valid which should not be.

The commitment/nullifier pattern

One powerful shielding pattern is to keep data in two different committed forms (referred to as "commitments" and "nullifiers"), with the former kept in a Merkle tree, and the latter in a `Set`. This lets us make single-use authentication tokens by first creating an entry in the Merkle tree and then when using it, proving its existence and adding the nullifier to a `Set`, asserting that it is not already there. This ensures that re-using the token isn't possible, while still not revealing which token was used. This is the underlying pattern of Zerocash and Zswap, which uses it to build shielded UTXOs.

It's crucial that the commitments and nullifiers use a domain separator to ensure they are not equal for the same secret data and, optionally, that creating the nullifier requires secret knowledge (as in the authenticating with hashes section), which ensu

Feedback

initial authorizer can't identify the token's use either.

Here's an example where public keys are authorized to increment a counter, once only:

```
import CompactStandardLibrary;

witness findAuthPath(pk: Bytes<32>): MerkleTreePath<10, Bytes<32>>;
witness secretKey(): Bytes<32>;

export ledger authorizedCommitments: HistoricMerkleTree<10, Bytes<32>>;
export ledger authorizedNullifiers: Set<Bytes<32>>;
export ledger restrictedCounter: Counter;

export circuit addAuthority(pk: Bytes<32>): [] {
  authorizedCommitments.insert(pk);
}

export circuit increment(): [] {
  const sk = secretKey();
  const authPath = findAuthPath(publicKey(sk));
  assert(authorizedCommitments.checkRoot(merkleTreePathRoot<10, Bytes<32>>(authPath)),
    "not authorized");
  const nul = nullifier(sk);
  assert !authorizedNullifiers.member(nul) "already incremented";
  authorizedNullifiers.insert(disclose(nul));
  restrictedCounter.increment(1);
}

circuit publicKey(sk: Bytes<32>): Bytes<32> {
  return persistentHash<Vector<2, Bytes<32>>>([pad(32, "commitment-domain"), sk]);
}
```

Feedback

```
circuit nullifier(sk: Bytes<32>): Bytes<32> {
  return persistentHash<Vector<2, Bytes<32>>>([pad(32, "nullifier-domain"), sk]);
}
```

Feedback