Writing a contract

Midnight comes with its own programming language, *Compact*, which enables you to write smart contracts as described in the overview. The Compact compiler outputs zero-knowledge circuits that are used to prove the correctness of interactions with the *ledger*. This page walks through the construction of a simple smart contract, which manages a publicly accessible value and supports the operations `get`, `set`, and `clear`. Because the value is public, anyone can call `get`, but if the value is currently set, only the user that last called `set` can `clear` it, and it must be `clear`ed before `set`ting it again.

To start, the contract specifies the version of the language it is using, imports Midnight's standard libraries, and declares an `enum` for the state it may currently be in:

```
pragma language_version 0.16;

import CompactStandardLibrary;

enum State {
  UNSET,
  SET
}
```

In addition to the `enum` declaration, custom data can also be defined with `struct`s. Details can be found in the language refe

Ask AI

The `ledger` section

Feedback

A key part of a Compact smart contract is the `ledger` section, which describes the state kept on chain. This example stores a key that can be used to identify the user who is permitted to `clear` the value, the value itself (this example uses a 64-bit unsigned integer), and what state the contract is in.

In addition to these, it is necessary to add a `round` counter, which is useful to retain anonymity, as discussed below.

Each field in a smart contract's on-chain state is declared with a `ledger` declaration. A constructor can be used to initialize the ledger fields when the contract is deployed. In this case, the ledger declarations are as follows:

```
export ledger authority: Bytes<32>;

export ledger value: Uint<64>;

export ledger state: State;

export ledger round: Counter;

constructor(sk: Bytes<32>, v: Uint<64>) {
  authority = disclose(publicKey(round, sk));
  value = disclose(v);
  state = State.SET;
}

circuit publicKey(round: Field, sk: Bytes<32>): Bytes<32> {
  return persistentHash<Vector<3, Bytes<32>>>(
           [pad(32, "midnight:examples:lock:pk"), round as Bytes<32>, sk]);
}
```

In addition to the `ledger` section, the `constructor` also demonstrates basic interaction with the state contained in it, using the field names to refer to the items in the ledger's state. Many ledger types also support *operations* as demonstrated in `clear`. Feedback

The `circuit` definitions

The example above already demonstrates use of a `circuit` to calculate the `publicKey` of a user. A `circuit` in Compact is equivalent to a function in many programming languages, but it is restricted to fixed computational bounds at compile time. A smart contract's `circuit`s can also be its main entry points; they are what users can call directly in transactions. Of the three entry points mentioned above, `get` is *unrestricted* and is simply implemented as follows:

```
export circuit get(): Uint<64> {
  assert(state == State.SET, "Attempted to get uninitialized value");
  return value;
}
```

Here `export` marks this circuit as an entry point to the smart contract, and `assert` ensures that it can only be used when the contract is in the correct state. The language reference details permissible contents of `circuit`s.

Local state and computations

The third context mentioned was the local machine of the user. This is explicitly programmable in the form of the DApp running on the user's machine. Compact can 'call out' to the local context through *witnesses*[^1], which are declared in a similar way to circuits. In this case, retrieving a user's *secret key* requires such a witness, because the secret must be kept local to a user's machine.

The code for this follows:

```
witness secretKey(): Bytes<32>;

export circuit set(v: Uint<64>): [] {
  assert(state == State.UNSET, "Attempted to set initialized value");
```

Feedback

```
    const sk = secretKey();
    const pk = publicKey(round, sk);
    authority = disclose(pk);
    value = disclose(v);
    state = State.SET;
  }

  export circuit clear(): [] {
    assert(state == State.SET, "Attempted to clear uninitialized value");
    const sk = secretKey();
    const pk = publicKey(round, sk);
    assert(authority == pk, "Attempted to clear without authorization");
    state = State.UNSET;
    round.increment(1);
  }
```

Note that the `witness` is not implemented in the Compact source code. Instead, the implementation is the responsibility of the TypeScript code of the DApp. It's important to note that each user could reasonably use a different implementation for the `witness`, so its results cannot be trusted inherently by the contract.

The full contract

All put together, the full example is:

```
pragma language_version 0.16;

import CompactStandardLibrary;

enum State {
```

Feedback

```
  UNSET,
  SET
}

export ledger authority: Bytes<32>;

export ledger value: Uint<64>;

export ledger state: State;

export ledger round: Counter;

constructor(sk: Bytes<32>, v: Uint<64>) {
  authority = disclose(publicKey(round, sk));
  value = disclose(v);
  state = State.SET;
}

circuit publicKey(round: Field, sk: Bytes<32>): Bytes<32> {
  return persistentHash<Vector<3, Bytes<32>>>(
           [pad(32, "midnight:examples:lock:pk"), round as Bytes<32>, sk]);
}

export circuit get(): Uint<64> {
  assert(state == State.SET, "Attempted to get uninitialized value");
  return value;
}

witness secretKey(): Bytes<32>;

export circuit set(v: Uint<64>): [] {
  assert(state == State.UNSET, "Attempted to set initialized value");
  const sk = secretKey();
```

Feedback

```
    const pk = publicKey(round, sk);
    authority = disclose(pk);
    value = disclose(v);
    state = State.SET;
  }

  export circuit clear(): [] {
    assert(state == State.SET, "Attempted to clear uninitialized value");
    const sk = secretKey();
    const pk = publicKey(round, sk);
    assert(authority == pk, "Attempted to clear without authorization");
    state = State.UNSET;
    round.increment(1);
  }
```

Basic confidentiality

It may not be immediately apparent what is held confidential in this example and what is enforced in the contract. Thankfully, both are well-defined:

- all data that is not a ledger field and is not an argument or return value of a ledger operation is kept confidential
- all computation that is not done in a `witness` function is enforced to be correct.

In particular, observe that this keeps the `secretKey` output confidential, while enforcing that *its hash* is the correct value in the case of `clear`.

This is also the reason for the `round` parameter: The `pk` "public key" *isn't* confidential, and would allow linkability between the same user publishing data in multiple rounds. By adding a round parameter into the public key computation, this linkability is br

Feedback

Despite the terms "secret key" and "public key", these two keys are *not* public key cryptography: they are simply a binary string and its hash. This is due to zero-knowledge circuits being able to have similar effects to digital signatures, relying only on the preimage resistance of hashes.

This pattern of hashing an arbitrary binary string and using it as a key is quite powerful. A similar concept that can be very useful is the use of *commitment schemes*, where arbitrary data is hashed together with a random *nonce*. The result can be safely placed into the ledger's state, without revealing the original data. (Note that the nonce *must* not be reused. If it is, you can link the commitments with the same nonces and values.) At a later point, the commitment can be "opened" by revealing the value and nonce, or a contract can simply prove (`assert`) that it *has* the correct value and nonce, without ever revealing them.

The `CompactStandardLibrary` module provides the following circuits for such uses:

```
circuit transientHash<T>(value: T): Field;
circuit transientCommit<T>(value: T, rand: Field): Field;
circuit persistentHash<T>(value: T): Bytes<32>;
circuit persistentCommit<T>(value: T, rand: Bytes<32>): Bytes<32>;
```

The `*Hash` variants are the basic hash function, with `*Commit` being a commitment function to arbitrary data. The `transient*` functions should only be used when the values are *not* kept in state, while `persistent*` outputs being suitable for storage in a contract's `ledger` state.

Next steps

This section continues with a more detailed overview of the Compact language. Alternatively, you may wish to jump to a more detailed example that showcases some more interesting things you can do with a DApp on Midnight. While this section has focused on the Compact language, the section about how Midnight works provides more detail about Midnight's ledger and how it functions

Feedback

[^1] The name *witness* comes from zero-knowledge literature; the etymology is roughly that it's the evidence you need to believe a statement. In this example, it's the evidence you need to believe that a `clear` was permissible.

Feedback