Smart contracts on Midnight

While you may have some familiarity with smart contracts, designing smart contracts for data protection provides unique challenges and perspectives. This article will therefore briefly walk through key points in which Midnight differs from more public smart contract solutions and how this should inform your construction of contracts in Midnight.

Replicated state machines

All blockchain systems are replicated state machines at their core: They keep a *ledger state*, which is modified by *transactions*. Various blockchains differ in which transactions are considered valid and what effect they have on a ledger state.

Smart-contract-enabled blockchains allow transactions to program parts of the blockchain's validity criteria that subsequent transactions have to satisfy. The focus here is on the *account model*, where contracts are *deployed* by a transaction, which assigns a unique address to the contract in the blockchain. This contract can define the validation criteria and state transitions for the transactions that interact with it.

The following example illustrates these ideas. Imagine a contract to support a guessing game, in which the player guesses factors of a number, stored in the contract's state. Making a correct guess allows the player to set the next number for the opposing player when making a guess, the player offers two factors for the current number and two factors to define the new number. The logi guess is expressed in pseudocode as follows:

> NOTE
>
> This is pseudocode, not a functional Compact program.

```
def guess_number(guess_a, guess_b, new_a, new_b):
  assert(guess_a != 1 and guess_b != 1 and new_a != 1 and new_b != 1,
    "1 is too boring a factor")
  assert(guess_a * guess_b == number,
    "Guessed factors must be correct")
  number = new_a * new_b
```

The contract *could* just let the player provide the new number directly, instead of its factors, but then they could (whether by accident or intentionally) also spoil the fun by passing in a prime number. Forcing the player to provide 'interesting' factors eliminates this possibility.

When the contract is deployed, this program is put directly on-chain, typically in a compressed, bytecode form, along with an initial state of the contract. Conceptually, this may make the ledger's state look something like this:

```
contracts:
  "<contract address>":
    state:
      number: 35
    entryPoints:
      guess_number: |
        def guess_number(...):
          // ...
```

A transaction can then call this contract by supplying inputs to the function, for instance:

```
transaction:
  type: "call"
  address: "<contract address>"
```

```
  entryPoint: "guess_number"
  inputs: [5, 7, 2, 6]
```

When processed, nodes process this by:

- looking up the `state` at `<contract address>` as well as the program at `<contract address>` and `guess_number`
- running the program against the state, and `inputs`
- if the program succeeds, storing the new `state`.

Midnight contracts, conceptually

You may have noticed that the above program is not a good implementation for this game, because every time a new number is set, its factors are publicly visible as part of the transaction that sets it. Anyone who really wants to win can read off the factors and use them as their own 'guess'. Where's the sport in that?

To move beyond this problem, imagine that you don't have to worry about the blockchain and how it processes transactions. Instead, consider a contract as an interactive program that can interact with the contract's on-chain state, as well as call arbitrary code on the user's local machine.

In this setting, it's possible to rewrite the above pseudocode program to look something like this:

```
def guess_number():
  (a, b) = local.guess_factors(number)
  assert(a != 1 and b != 1, "1 is too boring a factor")
  assert(a * b == number, "Guessed factors must be correct")
  (a, b) = local.new_challenge()
```

Feedback

```
    assert(a != 1 and b != 1, "1 is too boring a factor")
    number = a * b
```

While this program is longer than the one in the previous section, it is also doing slightly more. It tells us where the numbers come from: local calls to `guess_factors` or `new_challenge` respectively. Often this is what happens anyway, with transaction inputs needing to be carefully computed ahead of time to ensure that the corresponding transaction succeeds. Here, the API is clear, and the `guess_factors` routine is even given the number for which it should guess (which previously you'd have to figure out for yourself).

On the chain, this interaction would have the following interactions:

- retrieving `number` ledger field
- setting `number` ledger field.

Neither of these reveals the details of the factors - neither the ones guessed, nor the ones the new challenge consists of.

A practical challenge with this approach is how to ensure that the contract is correctly used. For the `local` calls, this is an accepted risk; we don't want to prescribe how `guess_factors` works, for instance, just that it outputs correct guesses (hence the input validation). For the contract program itself, we want other users to be convinced that we ran the right program and that the changes made to the contract's state are sensible.

Transcripts and ZK Snarks

The key technology that makes everything work is the ZK Snark. At their core, ZK Snarks (and more broadly zero-knowledge proofs) are a way to prove that you know how to assign values to a number of variables, so that they satisfy some set of clear, mathematical conditions. Some of these variables are *public*, while most are not.

The above program can be cleanly split into three interacting parts, each run in a separate environment: The `local` part, the `ledger` part, and the glue that links the two together and encodes the core program logic. This 'glue' can be converted into a series of variable assignments and equations that can be transformed into a ZK Snark, while the ledger interactions can be converted into a program that runs on-chain.

Here, in the example of factoring the current state of `35` into `5 * 7` and replacing it with `2 * 6`:

Off-chain code

```
(a, b) = local.
  guess_factors(n1)


(a, b) = local.
  new_challenge()
```

In-circuit code

```
def guess_number():


  assert(a != 1 and b != 1,
"...")


  assert(a * b == n2, "...")


  assert(a != 1 and b != 1,
"...")

  n3 = a * b
```

On-chain code

```
n1 = number


n2 = number


number = n3
```

Private transcript

Public transcript

```
a1 = 5
b1 = 7
a2 = 2
b2 = 6
```

Circuit constraints

```
guess_number:
  inputs:
    public: n1, n2, n3,
      transcript code
    private: a1, b1, a2, b2
  constraints:
    a1 != 1
    b1 != 1
    a1 * b1 = n2
    a2 != 1
    b2 != 1
    n3 = a2 * b2
    // Additional constraints
    // enforcing the shape of
    // the public transcript
```

```
n1 = 35
n2 = 35
n3 = 12

assert(n1 == number)
assert(n2 == number)
number = n3
```

More complex programs, with function calls, conditionals, iterations, and complex primitives such as hash function calls can also be translated in this way. See the writing a contract section of this documentation for a description of the language we use to write these programs.

In the preceding example, it is possible to prove that, for public `n1`, `n2` and `n3`, we know values of `a1`, `b1`, `a2`, and `b2` for which these equations hold. This proof does not say that anyone actually ran the above program, but it *does* say that the program's rules were followed, which is what a skeptical user truly cares about.

The sequence of assignments, `n1`, `n2`, and `n3`, and the program that produces or uses them is referred to as the *public transcript*, and conversely `a1`, `b1`, `a2`, and `b2` are the *private transcript*. Public transcripts are encoded as bytecode[^1], and the shape of this bytecode is directly enforced by the circuit.

Feedback

Transactions in Midnight then are essentially made up of the public transcript and a zero-knowledge proof that this transcript is correct. Each transaction is made with respect to a contract and a specific *circuit*[^2] on that contract. On-chain, instead of storing the code for `guess_number()`, a cryptographic key used to verify zero-knowledge proofs *for* `guess_number()` is stored. This cryptographically encodes and enforces all of the equations listed in the circuit above.

Broadly, the state looks something like this:

```
contracts:
  "<contract address>":
    state:
      number: 35
    entryPoints:
      guess_number: "<verifier key>"
```

And a transaction made against this state might look something like:

> NOTE
>
> This is a sketch of a transaction.

```
transaction:
  type: "call"
  address: "<contract address>"
  entryPoint: "guess_number"
  transcript: |
    n1 = 35
    n2 = 35
    n3 = 12
    assert(n1 == number)
```

Feedback

```
    assert(n2 == number)
    number = n3
proof: "<zero-knowledge proof>"
```

This transaction, when it is verified, will check that the proof is valid with respect to the verifier key and then *run* the transcript. Here it checks that things are still as expected; if the current `number` *isn't* `35` the transaction is no longer valid – whoever made it did not guess the factors of 35, after all. The result of the transaction (if it succeeds) is updating the state to contain `12` – and importantly, *this* transaction tells no one which factors were used there, or in the guess!

A reasonable question is why the `number` check occurs twice, and in practice, this is a valid observation: There is no need to read the same value multiple times. However, this way of handling external interactions means that the operations performed here are arbitrary; the zero-knowledge proof had no knowledge of what a `read` is or that the values of `n1` and `n2` are necessarily the same, and this allows the use of more interesting operations, such as `increment` or `insert`. These are particularly useful to avoid making transactions invalid due to results not matching, as in the case of `35` above. Contrast two simultaneous invocations of `increment` with two simultaneous sequences of `read`ing a value, adding `1`, and `write`ing it again; the `increment` will (almost) always succeed, while the read-add-write sequence is prone to failure.

Putting value at stake

It's not immediately obvious how the notion of *value* fits into this model. In public blockchains, it's easy for a smart contract to have a value as well as a state, which can be used to pay into and out of the contract. As this transfer of value is important to many applications, it is necessary to achieve such transfers in a setting that preserves data privacy.

The Midnight token currently uses an implementation of Zswap, which operates similarly to UTXOs, but shields the token values, types, and fund holders. An exception to total shielding applies to the funds held by a contract; the value and type of these are still shielded by default, but holding and releasing them *is* linked to the contract.

Feedback

These UTXOs are represented in contracts as individual coins, which are just data until they are explicitly *received*. Once received, they can be handled like any other data – whether they are stored publicly, encrypted, or stored privately is up to the contract itself. When a contract wishes, they can then be *sent* to another contract or to a user address.

Coin *receives* and *sends* have special semantics: They are recorded as operations in the public transcript but have no effect on the contract's state. Instead, they require a corresponding input or output to be included in the same transaction, ensuring that a contract doesn't receive funds that don't exist or send funds it doesn't have.

Again in pseudocode, wagers can be attached to the example:

```
def guess_number(new_wager):
  (a, b) = local.guess_factors(number)
  builtin.send(wager, local.self())
  assert(a != 1 and b != 1, "1 is too boring a factor")
  assert(a * b == number, "Guessed factors must be correct")
  (a, b) = local.new_challenge()
  assert(a != 1 and b != 1, "1 is too boring a factor")
  number = a * b
  builtin.receive(new_wager)
  wager = new_wager
```

Factoring and keys

The example of factoring may seem like a toy, and it is somewhat arbitrary, but it is worth noting that factoring large integers is an important problem in cryptography. Knowing the factors of large numbers is the basis of the RSA cryptographic algorithms, and the simple guessing game corresponds to proving that you know the secret keys for an RSA public key. This shows the power of zero-knowledge proofs and that they can serve the same purposes as signature schemes. Not only can you prove you know a secret key, but you can then prove that *the same person* did something else, effectively signing what they did.

In practice, if you want to authenticate, this construction is not the most efficient; proving the knowledge of a preimage of hash functions (that is, knowing `sk` such that `pk = H(sk)`) is a simpler alternative in most cases.

[^1] For advanced reading of *how* the operations are encoded, see the details of Midnight's on-chain VM, Impact.

[^2] Circuits are named such as the compilation of zero-knowledge proofs has many similarities with assembling a special-purpose logic circuit