Explicit disclosure in Compact: The Midnight "Witness Protection Program"

Introduction

Midnight supports the development of applications that preserve privacy whenever possible while allowing selective disclosure of private information as necessary. Selective disclosure differs from traditional blockchains, in which everything is public, or strict privacy-preserving blockchains, in which everything is private. Midnight's selective disclosure allows banks, for example, to disclose data required for regulatory purposes while keeping other account information private.

The decision to disclose private information, including information derived from private information, must rest with each Midnight DApp because disclosure requirements are inherently situation-specific. However, because private information should be disclosed only as necessary, Midnight's Compact language requires disclosure to be explicitly declared. That is, a Compact program must explicitly declare its intention to disclose data that might be private before storing it in the public ledger, returning it from an exported circuit, or passing it to another contract. This makes privacy the default and disclosure an explicit exception, reducing the risk of accidental disclosure.

The contract produced from a Compact program is a zero-knowledge (zk) proof coupled with updates to be made to the publi
A zk-proof proves some property about one or more pieces of data, referred to as *witnesses* or *witness data*, without disclosing anything about the witness data except that the property holds for them. In Compact, witness data comes primarily from external callback functions declared as witnesses in a Compact contract and provided to the contract by a DApp. Witness data can also co
into a contract via exported circuit arguments and via the arguments to the contract constructor, if any. Also, any value

witness data is also considered to be witness data. Because witness data may contain private information, it should ordinarily be used only for constructing the zk-proof and not disclosed in any way, but exceptions sometimes need to be made. When exceptions are made, the disclosure must be explicitly declared.

Explicitly declaring disclosure

Explicitly declaring intent to disclose witness data in Compact is straightforward, requiring only the addition of a `disclose()` wrapper around any expression whose value may contain witness data to be disclosed, as illustrated by the following trivial program:

```
import CompactStandardLibrary;
witness getBalance(): Bytes<32>;
export ledger balance: Bytes<32>;

export circuit recordRalance(): [] {
  balance = disclose(getBalance());
}
```

Without the `disclose()` wrapper, the compiler rejects the program with an informative error message. For example, attempting to compile this Compact program:

```
import CompactStandardLibrary;
witness getBalance(): Bytes<32>;
export ledger balance: Bytes<32>;

export circuit recordBalance(): [] {
  balance = getBalance();       // missing disclose() wrapper
}
```

Feedback

causes the compiler to abort with the following error message:

```
Exception: /tmp/q3.compact line 6 char 11:
  potential witness-value disclosure must be declared but is not:
    witness value potentially disclosed:
      the return value of witness getBalance at line 2 char 1
    nature of the disclosure:
      ledger operation might disclose the witness value
    via this path through the program:
      the right-hand side of = at line 6 char 11
```

The error message lists the sources of all witness data disclosed at this point, so the programmer knows that adding a `disclose()` wrapper will declare all of them disclosed.

Placing a `disclose()` wrapper does not cause disclosure in itself; in fact, it has no effect other than telling the compiler that it is okay to disclose the value of the wrapped expression. Equivalently, it tells the compiler to pretend that the value of the wrapped expression does not contain witness data whether it actually does or not.

Following indirect witness data assignments

In many cases, disclosure is not so direct, yet the requirement to explicitly declare disclosure always holds. For example, obfuscating the disclosure as follows:

```
import CompactStandardLibrary;
struct S { x: Field; }
witness getBalance(): Bytes<32>;
export ledger balance: Bytes<32>;
```

```
circuit obfuscate(x: Field): Field { // seriously bad obfuscation
  return x + 73;
}

export circuit recordBalance(): [] {
  const s = S { x: getBalance() as Field };
  const x = obfuscate(s.x);
  balance = x as Bytes<32>;
}
```

still causes the compiler to abort, with a similar error message:

```
Exception: /tmp/q3.compact line 13 char 11:
  potential witness-value disclosure must be declared but is not:
    witness value potentially disclosed:
      the return value of witness getBalance at line 3 char 1
    nature of the disclosure:
      ledger operation might disclose the result of an addition involving the witness value
    via this path through the program:
      the binding of s at line 11 char 3
      the argument to obfuscate at line 12 char 13
      the computation at line 7 char 10
      the binding of x at line 12 char 3
      the right-hand side of = at line 13 char 11
```

In this case, the disclosure, if intentional, can be declared by placing a `disclose()` wrapper around the call to `getBalance()`, around the right-hand-side of the `balance` assignment, or anywhere else along the path from the point of call to the point of disclosure. For example, it could be added to the body of the `obfuscate` circuit around the reference to its argument.

```
import CompactStandardLibrary;
struct S { x: Field; }
witness getBalance(): Bytes<32>;
export ledger balance: Bytes<32>;

circuit obfuscate(x: Field): Field { // seriously bad obfuscation
  return disclose(x) + 73;
}

export circuit recordBalance(): [] {
  const s = S { x: getBalance() as Field };
  const x = obfuscate(s.x);
  balance = x as Bytes<32>;
}
```

The best practice is most often to put the `disclose()` wrapper as close to the disclosure point as possible to avoid accidental disclosure if the data travels along multiple paths. For a structured value (such as a tuple, vector, or struct), however, the `disclose()` wrapper should be placed only around the portions of the value that are expected to contain witness data to avoid accidental disclosure in the other portions of the value. Also, in the case of a witness that always returns non-private data or private data that has been sufficiently obfuscated via some cryptographically sound mechanism, it makes sense to place the `disclose()` wrapper directly on the call to the witness.

Indirect disclosure via conditional expression

The preceding example illustrates that subjecting witness data to arithmetic, converting it from one representation to another, and passing it into and out of other circuits does not hide potential disclosure from the compiler. The compiler also detects indirect disclosure via conditional expressions. For example:

Feedback

```
import CompactStandardLibrary;
witness getBalance(): Uint<64>;

export circuit balanceExceeds(n: Uint<64>): Boolean {
  return getBalance() > n;
}
```

causes the compiler to abort with the following message:

```
Exception: /tmp/q3.compact line 5 char 3:
  potential witness-value disclosure must be declared but is not:
    witness value potentially disclosed:
      the return value of witness getBalance at line 2 char 1
    nature of the disclosure:
      the value returned from exported circuit balanceExceeds might disclose the result of a
      comparison involving the witness value
    via this path through the program:
      the comparison at line 5 char 10
```

The message aids the programmer by noting the indirect nature of the disclosure. This example also illustrates that disclosure does not occur just when witness data is stored in the ledger but also when returned from an exported circuit.

Safe Compact standard library routines

The compiler recognizes that certain Compact standard library routines sufficiently disguise witness data so that explicit declaration of disclosure is not required. For an expression `e` whose value contains witness data, the compiler will treat `transientCommit(e)` as if it does not contain witness data, while it will treat `transientHash(e)` as if it does.

How explicit disclosure is implemented

We refer to the portion of the compiler that detects and reports undeclared disclosure of witness data as the "witness protection program". The witness-protection program is implemented as an *abstract interpreter*, where the abstract values are not actual run-time values but information about witness data that will be contained within the actual run-time values.

The abstract interpreter evaluates the program as if the abstract values were actual values. The operations performed by the interpreter are modified, however, to propagate (or not) information about witness data from the operation inputs to the operation outputs. If at some point the interpreter encounters an undeclared disclosure, e.g., a ledger store, of an abstract value containing witness data, the compiler halts and produces an appropriate error message.

Conclusion

The `disclose()` wrapper in Compact enforces deliberate programming decisions when dealing with potentially sensitive private witness data and data derived from private information. Explicit disclosure requires Compact program assignments to declare an intention to use data that might be private before storing it in the public ledger, returning it from an exported circuit, or passing it to another contract. This makes privacy the default and disclosure an explicit exception, reducing the risk of accidental disclosure.