Confidential Transactions--
Principal Investigator: Greg Maxwell

One of the most powerful new features being explored in Elements[*] is
Confidential Transactions, a cryptographic tool to improve the privacy and
security of Bitcoin. This feature keeps the amounts transferred visible
only to participants in the transaction (and those they designate).

[*] https://blockstream.com/developers/

The security of the Bitcoin ledger is made possible by universal
verification: each participant individually and autonomously verifies
that each transaction is valid, without trusting any third party.
An unfortunate side effect is that all the transaction data must be
conspicuously public so it can be verified, which is at odds with the
normal expectation of privacy for traditional monetary instruments.

Insufficient financial privacy can have serious security and privacy
implications for both commercial and personal transactions. Without
adequate protection, thieves and scammers can focus their efforts on
known high-value targets, competitors can learn business details, and
negotiating positions can be undermined. Since publishing often requires
spending money, lack of privacy can chill free speech. Insufficient
privacy can also result in a loss of fungibility--where some coins are
treated as more acceptable than others--which would further undermine
Bitcoin’s utility as money.

Bitcoin partially addresses the privacy problem by using pseudonymous
addresses. If someone does not know which users own which addresses,
the privacy impact is reduced. But any time you transact with someone
you learn at least one of their addresses. From there, you can trace out
other connected addresses and estimate the values of their transactions
and holdings. For example, suppose your employer pays you with Bitcoin
and you later spend those coins on your rent and groceries. Your landlord
and the supermarket will both learn your income, and could charge you
higher prices as your income changes or target you for theft.

There are existing deployed techniques that further improve privacy
in Bitcoin (such as CoinJoin, which merges the transaction history of
users by making joint payments), but the utility of these techniques is
reduced by the fact that it’s possible to track amounts.

There have been proposed cryptographic techniques to improve privacy
in Bitcoin-like systems, but so far all of them result in breaking
"pruning" (section 7 of Bitcoin.pdf) and result in participants
needing a perpetually growing database to verify new transactions,
because these systems prevent learning which coins have been spent. Most
proposed cryptographic privacy systems also have poor performance, high
overhead, and/or require new and very strong (and less well understood)
cryptographic assumptions.

Confidential Transactions improves the situation by making the transaction
amounts private, while preserving the ability of the public network
to verify that the ledger entries still add up. It does this without
adding any new basic cryptographic assumptions to the Bitcoin system,
and with a manageable level of overhead.

CT is possible due to the cryptographic technique of additively
homomorphic commitments. As a side-effect of its design, CT also enables
the additional exchange of private "memo" data (such as invoice numbers
or refund addresses) without any further increase in transaction size,
by reclaiming most of the overhead of the CT cryptographic proofs.

The technology behind Confidential Transactions
A high level technical primer

This work was originally proposed by Adam Back on
Bitcointalk in his 2013 thread "bitcoins with homomorphic value"

[https://bitcointalk.org/index.php?topic=305791.0]. To build CT I had to implement several new cryptosystems which work in concert, and invented a generalization of ring signatures and several novel optimizations to make the result reasonably efficient.

The basic tool that CT is based on is a Pedersen commitment.

A commitment scheme lets you keep a piece of data secret but commit to it so that you cannot change it later. A simple commitment scheme can be constructed using a cryptographic hash:

```
  commitment = SHA256( blinding_factor || data )
```

If you tell someone only the commitment then they cannot determine what data you are committing to (given certain assumptions about the properties of the hash), but you can later reveal both the data and the blinding factor and they can run the hash and verify that the data you committed to matches. The blinding factor is present because without one, someone could try guessing at the data; if your data is small and simple, it might be easy to just guess it and compare the guess to the commitment.

A Pedersen commitment works like the above but with an additional property: commitments can be added, and the sum of a set of commitments is the same as a commitment to the sum of the data (with a blinding key set as the sum of the blinding keys):

```
  C(BF1, data1) + C(BF2, data2) == C(BF1 + BF2, data1 + data2)
  C(BF1, data1) - C(BF1, data1) == 0
```

In other words, the commitment preserves addition and the commutative property applies.

If data_n = {1,1,2} and BF_n = {5,10,15} then:

```
  C(BF1, data1) + C(BF2, data2) - C(BF3, data3) == 0
```

and so on.

Our specific Pedersen commitments are constructed using elliptic curve points. [The reader need not understand elliptic curve cryptography, beyond accepting the black box behaviors I describe here.]

Normally an ECC pubkey is created by multiplying a generator for the group (G) with the secret key (x):

```
  Pub = xG
```

The result is usually serialized as a 33-byte array.

ECC public keys obey the additively homomorphic property mentioned before:

```
   Pub1 + Pub2 = (x1 + x2 (mod n))G.
```

(This fact is used by the BIP32 HD wallet scheme to allow third parties to generate fresh Bitcoin addresses for people.)

The Pedersen commitment is created by picking an additional generator for the group (which we'll call H) such that no one knows the discrete log for H with respect to G (or vice versa), meaning no one knows an x such that xG = H. We can accomplish this by using the cryptographic hash of G to pick H:

```
    H = to_point(SHA256(ENCODE(G)))
```

Where to_point() takes the input as the x value of a new point, checks that its on the curve and solves for the y value.

Given our two generators we can build a commitment scheme like this:

```
   commitment = xG + aH
```

Here x is our secret blinding factor, and a is the amount that we're committing to. You can verify just using the commutative property of addition that all the relationships given for an additively homomorphic commitment scheme hold.

The Pedersen commitments are information-theoretically private: for any commitment you see, there exists some blinding factor which would make any amount match the commitment. Even an attacker with infinite computing power could not tell what amount you committed to, if your blinding factor was truly random. They are computationally secure against fake commitment, in that you can't actually compute that arbitrary mapping; if you can it means you can find the discrete log of the generators with respect to each other, which means that the security of the group is compromised.

With this tool in hand we can go and replace the normal 8-byte integer amounts in Bitcoin transactions with 33-byte Pedersen commitments.

If the author of a transaction takes care in picking their blinding factors so that they add up correctly, then the network can still verify the transaction by checking that its commitments add up to zero:

```
    (In1 + In2 + In3 + plaintext_input_amount*H...) -
     (Out1 + Out2 + Out3 + ... fees*H) == 0.
```

This requires making the fees in a transaction explicit, but that's generally desirable.

The commitment and its checking are quite simple. Unfortunately, without additional measures this scheme is insecure.

The problem is that the group is cyclic, and addition is mod P (a 256-bit prime number that defines the order of the group). As a result, addition of large values can 'overflow' and behave like negative amounts. This means that a sums-to-zero behavior still holds when some outputs are negative, effectively allowing the creation of 5 coins from nothing:

```
 (1 + 1) - (-5 + 7) == 0
```

This would be interpreted as "someone spends two bitcoins, gets a '-5' bitcoin out that they discard out, and a 7 bitcoin output".

In order to prevent this, when there are multiple outputs we must prove that each committed output is within a range which cannot overflow (e.g. [0, 2^64)).

We could just disclose the amounts and blinding factors so that the network could check, but this would lose all of the privacy. So, instead, we need to prove that a committed amount is within the range but reveal nothing else about it: we need an additional cryptosystem to prove the range of a Pedersen commitment. We use a scheme similar to Schoenmakersâ\200\231 binary decomposition but with many optimizations (including not using binary).

To build this we start with a basic EC signature. If a signature is constructed so that the 'message' is the hash of the pubkey, the signature proves that the signer knew the private key, which is the discrete log of the pubkey with respect to some generator.

For a 'pubkey' like P = xG + aH, no one knows the discrete log of P with respect to G because of the addition of H, because no one knows an x for xG = H----_unless_ a is 0. If a is zero then P = xG and the discrete log is just x; someone could sign for that pubkey.

A pedersen commitment can be proven to be a commitment to a zero by just signing a hash of the commitment with the commitment as the public key. Using the public key in the signature is required to prevent setting the signature to arbitrary values and solving for the commitment. The private key used for the signature is just the blinding factor.

Going further, letâ\200\231s say I want to prove C is a commitment to 1 without telling you the blinding factor. All you do is compute

    C' = C - 1H

and ask me to provide a signature (with respect to G) with pubkey C'. If
I can do that, the C must be a commitment to 1 (or else I've broken the
EC discrete log security).

To avoid giving away the amount we need yet another cryptographic
construct: a ring signature. A ring signature is a signature scheme
where there are two (or more) pubkeys and the signature proves that the
signer knows the discrete log of at least one of the pubkeys.

So with that we can construct a scheme where I prove a commitment that
C is either 0 or 1--we call this an "OR proof".

First, I give you C, and you compute C':

    C' = C - 1H

Then I provide a ring signature over {C, C'}.

If C was a commitment to 1 then I do not know its discrete log, but
C' becomes a commitment to 0 and I do know its discrete log (just the
blinding factor). If C was a commitment to 0 I know its discrete log,
and I don't for C'. If it was a commitment to any other amount, none
of the result will be zero and I won't be able to sign.

This works for any pair of numbers, just by suitably pre-processing the
amounts that are put into the ring... or even for more than two numbers.

Say I want to prove to you that C is in the range [0, 32). Now that we
have an OR proof, imagine I send you a collection of commitments and OR
proofs for each of them:

C1 is 0 or 1 C2 is 0 or 2 C3 is 0 or 4 C4 is 0 or 8 C5 is 0 or 16.

If I pick the blinding factors for C1..5 correctly then I can arrange
it so that C1 + C2 + C3 + C4 + C5 == C. Effectively I have built up
the number in binary, and a 5-bit number can only be in the range [0,32).

Numerous optimizations are required to make this more efficient:

First, I propose a new ring
signature formulation, a Borromean ring signature[*], which is especially
efficient: it requires only 32 bytes per pubkey, plus 32 bytes which
can be shared by many separate rings. This is has twice the asymptotic
efficiency of previously proposed constructions for this application.

[*] https://github.com/Blockstream/borromean_paper/raw/master/borromean_draft_0.01_8c3f9e7.pdf

Instead of expressing the amount directly, CT amounts are expressed
using a decimal floating point where the digits are multiplied by a
base 10 exponent. This means that you can prove large amounts with
small proofs, so long as they have few significant digits in base 10:
e.g., 11.2345 and .0112345 can have the same size proof, even though
one number is a thousand times larger.

There is also a non-private "minimum amount" sent, which allows a smaller
proof to cover a larger range if the user doesn't mind leaking some
information about the minimum amount (which might already be public for
external reasons); this also allows the least significant digits to be
non-zero when an exponent is used. Minimum amounts are supported by first
subtracting the minimum, then proving that the result is non-negative.

The mantissa of the floating point is encoded using rings of size 4 (base
4) rather than binary, because this minimizes the number of commitments
sent while not using any more signature data than base 2.

The final mantissa digit commitment can be skipped, backwards constructing it from the value being proven and the other digits, etc.

Finally, by careful use of derandomized signing in the prover, it’s possible for the receiver of the coins--who shares a secret with the sender, due to ECDH key agreement with the receivers pubkey--to ’rewind’ the proof and use it to extract a message sent by the sender which is 80% of the size of the proof. We use this to signal the value and blinding factor to the receiver, but it could also be used to carry things like reference numbers or refund addresses.

The result is that a proof for a 32-bit value is 2564 bytes, and simultaneously may convey 2048 bytes of message. A 32-bit proof can cover a range of 42.94967296 BTC with 1e-8 precision, or 429.4967296 BTC with 1e-7 precision, and so on. My implementation is able to verify over 1300 32-bit range proofs per second on an i7-4770R, and there are many performance optimizations still possible.

The implementation supports proofs of any mantissa size or exponent, with the parameters controlled by the sender. Performance and size are linear in the number of mantissa bits, and odd numbers of bits are supported (by switching to radix-2 for the last digit).

In Elements, the range proofs are only required in cases where there are multiple confidential value outputs (including fees). Transactions that merge multiple confidential amounts into a single output do not need range proofs since the fact that all the inputs were in range is sufficient.

By sharing the scanning key used to establish the shared secret used by the rewindable range proofs, this approach is completely compatible with watching wallets; users can share these keys with auditors to enable them to view their transaction amounts.

Future work may use the fact that proofs can support a minimum value to also allow skipping the range proofs when there is a single confidential output even when fees are being paid, or allow nodes to skip or delay verifying most range proofs by using fraud proofs.

The system presented here depends on no new fundamental cryptographic assumptions, only the hardness of the discrete log problem in the secp256k1 group and a random oracle assumption, just like the normal signatures in Bitcoin.

While the size of the range proofs are non-trivial, they are still an order of magnitude smaller and faster to verify than some alternatives (like Zerocoin), and most of their space can be reclaimed to communicate additional data between users, a feature which is often requested but hard to justify in a public broadcast network. Similar to signatures, the range proofs can be placed on separate tree branches in blocks to allow clients that donâ\200\231t care about (e.g. historical ones) to skip receiving them.

Most importantly, this scheme is compatible with pruning and does not make the verification state for Bitcoin grow forever. It is also compatible with CoinJoin and CoinSwap, allowing for transaction graph privacy as well while simultaneously fixing the most severe limitation of these approaches to privacy (that transaction amounts compromise their privacy).

Unlike some other proposals, this system is not just speculation or pure cryptography without integration with the Bitcoin system.

The base cryptosystem is implemented in a fork of libsecp256k1, available at: https://github.com/ElementsProject/secp256k1-zkp

Confidential Transactions is enabled in Elements Alpha and used by default by for all ordinary transactions.