Contracts

This document describes smart contracts that can be setup using Grin even though the Grin chain does not support scripting. All these contracts rely on a few basic features that are built in the chain and compose them in increasingly clever ways.

None of those constructs are fully original or invented by the authors of this document or the Grin development team. Most of the credit should be attributed to a long list of cryptographers and researchers. To name just a few: Torben Pryds Pedersen, Gregory Maxwell, Andrew Poelstra, John Tromp, Claus Peter Schnorr. We apologize in advance for all those we couldn't name and recognize that most computer science discoveries are incremental.

Built-Ins

This section is meant as a reminder of some crucial features of the Grin chain. We assume some prior reading as to how these are constructed and used.

Pedersen Commitments

All outputs include a Pedersen commitment of the form `r*G + v*H` with `r` the blinding factor, `v` the value, and G and H two distinct generator points on the same curve group.

Aggregate Signatures (a.k.a. Schnorr, MuSig)

We suppose we have the SHA256 hash function and the same G curve as above. In its simplest form, an aggregate signature is built from:

- the message `M` to sign, in our case the transaction fee
- a private key `x`, with its matching public key `x*G`
- a nonce `k` just used for the purpose of building the signature

We build the challenge `e = SHA256(M | k*G | x*G)`, and the scalar `s = k + e * x`. The full aggregate signature is then the pair `(s, k*G)`.

The signature can be checked using the public key `x*G`, re-calculating `e` using M and `k*G` from the 2nd part of the signature pair and by verifying that `s`, the first part of the signature pair, satisfies:

```
s*G = k*G + e * x*G
```

In this simple case of someone sending a transaction to a receiver they trust (see later for the trustless case), an aggregate signature can be directly built for a Grin transaction by taking the above private key `x` to be the sum of output blinding factors minus the sum of input blinding factors. The resulting kernel is assembled from the aggregate signature generated using `r` and the public key `r*G`, and allows to verify non-inflation for all Grin transactions (and signs the fees).

Because these signatures are built simply from a scalar and a public key, they can be used to construct a

variety of contracts using "simple" arithmetic.

(Absolute) Timelocked Transactions

Analogous to Bitcoin nLockTime.

A transaction can be time-locked with a few simple modifications:

- the message `M` to sign becomes the lock_height `h` at which the transaction becomes spendable appended to the fee
  - `M = fee | h`
- the lock height `h` is included in the transaction kernel
- a block with a kernel that includes a lock height greater than the current block height is rejected

(Relative) Timelocked Transactions

We can extend the concept of an absolute locktime on a tx by including a (kernel) commitment that we can define the lock_height relative to.

The lock_height would be relative to the block height where the referenced kernel was first included in the chain state.

Tx2 can then be restricted such that it would only be valid to include it in a block onc`h` blocks have passed after first seeing Tx1 (via the referenced kernel commitment).

- the message `M` to sign would need to include the following -
  - the `fee` as before
  - the lock_height `h` (as before but interpreted as a relative value)
  - a referenced kernel commitment `C`
  - M = `fee | h | C`

For Tx2 to be accepted it would also need to include a Merkle proof identifying the block includin`C` from Tx1. This proves the relative lock_height requirement has been met.

Derived Contracts

Trustless Transactions

An aggregate (Schnorr) signature involving a single party is relatively simple but does not demonstrate the full flexibility of the construction. We show here how to generalize it for use in outputs involving multiple parties.

As constructed in section 1.2, an aggregate signature requires trusting the receiving party. As Grin outputs are completely obscured by Pedersen Commitments, one cannot prove money was actually sent to the right party, hence a receiver could claim not having received anything. To solve this issue, we require the receiver to collaborate with the sender in building a transaction and specifically its kernel signature.

Alice wants to pay Bob in grins. She starts the transaction building process:

1. Alice selects her inputs and builds her change output. The sum of all blinding factors (change output minus inputs) is `rs`.
2. Alice picks a random nonce ks and sends her partial transaction `ks*G` and `rs*G` to Bob.
3. Bob picks his own random nonce `kr` and the blinding factor for his output `rr`. Using `rr`, Bob adds his output to the transaction.
4. Bob computes the message `M = fee | lock_height`, the Schnorr challenge `e = SHA256(M | kr*G + ks*G | rr*G + rs*G)` and finally his side of the signature `sr = kr + e * rr`.
5. Bob sends `sr`, `kr*G` and `rr*G` to Alice.
6. Alice computes `e` just like Bob did and can check that `sr*G = kr*G + e*rr*G`.
7. Alice sends her side of the signature `ss = ks + e * rs` to Bob.
8. Bob validates `ss*G` just like Alice did for `sr*G` in step 6 and can produce the final signature `s = (ss + sr, ks*G + kr*G)` as well as the final transaction kernel including `s` and the public key `rr*G + rs*G`.

This protocol requires 3 data exchanges (Alice to Bob, Bob back to Alice, and finally Alice to Bob) and is therefore said to be interactive. However the interaction can be done over any medium and in any period of time, including the pony express over 2 weeks.

This protocol can also be generalized to any number `i` of parties. On the first round, all the `ki*G` and `ri*G` are shared. On the 2nd round, everyone can compute `e = SHA256(M | sum(ki*G) | sum(ri*G))` and their own signature `si`. Finally, a finalizing party can then gather all the partial signatures `si`, validate them and produce `s = (sum(si), sum(ki*G))`.

Multiparty Outputs (multisig)

We describe here a way to build a transaction with an output that can only be spent when multiple parties approve it. This construction is very similar to the previous setup for trustless transactions, however in this case both the signature and a Pedersen Commitment need to be aggregated.

This time, Alice wants to send funds such that both Bob and her need to agree to spend. Alice builds the transaction normally and adds the multiparty output such that:

1. Bob picks a blinding factor `rb` and sends `rb*G` to Alice.
2. Alice picks a blinding factor `ra` and builds the commitment `C = ra*G + rb*G + v*H`. She sends the commitment to Bob.
3. Bob creates a range proof for `v` using `C` and `rb` and sends it to Alice.
4. Alice generates her own range proof, aggregates it with Bob, finalizing the multiparty output `Oab`.
5. The kernel is built following the same procedure as for Trustless Transactions.

We observe that for that new output `Oab`, neither party know the whole blinding factor. To be able to build a transaction spending Oab, someone would need to know `ra + rb` to produce a kernel signature. To produce that spending kernel, Alice and Bob need to collaborate. This, again, is done using a protocol very close to Trustless Transactions.

Multiparty Timelocks

This contract is a building block for multiple other contracts. Here, Alice agrees to lock some funds to start a financial interaction with Bob and prove to Bob she has funds. The setup is the following:

- Alice builds a a 2-of-2 multiparty transaction with an output she shares with Bob, however she does not participate in building the kernel signature yet.
- Bob builds a refund transaction with Alice that sends the funds back to Alice using a timelock (for example 1440 blocks ahead, about 24h).
- Alice and Bob finish the 2-of-2 transaction by building the corresponding kernel and broadcast it.

Now Alice and Bob are free to build additional transactions distributing the funds locked in the 2-of-2 output in any way they see fit. If Bob refuses to cooperate, Alice just needs to broadcast her refund transaction after the time lock expires.

This contract can be trivially used for unidirectional payment channels.

Conditional Output Timelocks

Analogous to Bitcoin CheckLockTimeVerify.

We currently have *unconditional* lock_heights on txs (tx is not valid and will not be accepted until lock_height has passed).

Private keys can be summed together. $Key_3 = Key_1 + Key_2$

Commitments can be summed together. $C_3 = C_1 + C_2$

Given *unconditional locktimes on txs* we can leverage these to give us *conditional locktimes on outputs* by "entangling" two outputs on two related txs together.

We can construct two txs (Tx_1, Tx_2) with two entangled outputs Out_1 and Out_2 such that -

- Out_1 (commitment C_1) is from Tx_1 and built using Key_1
- Out_2 (commitment C_2) is from Tx_2 and built using Key_2
- Tx_2 has an *unconditional* lock_height on it

If we do this (and we can manage the keys as necessary) -

- $Out_1 + Out_2$ can *only* be spent as a pair using Key_3
- They can *only* be spent after lock_height from Tx_2

Tx_1 (containing Out_1) can be broadcast, accepted and confirmed on-chain immediately. Tx_2 cannot be broadcast and accepted until lock_height has passed.

So if Alice only knows K_3 and does not know Key_1 or Key_2, then Out_1 can only be spent by Alice after lock_height has passed. If Bob on the other hand knows Key_2 then Out_1 can be spent by Bob immediately.

We have a *conditional* timelock on Out_1 (confirmed, on-chain) where it can be spent either with Key_3 (after lock_height), *or* Key_2 immediately.

(Relative) Conditional Output Timelocks

Analogous to Bitcoin CheckSequenceVerify.

By combining "Conditional Timelock on Output" with "(Relative) Timelocked Transactions" we can encumber a confirmed output with a relative timelock (relative to a related tx kernel).

Tx_1 (containing Out_1) can be broadcast, accepted and confirmed on-chain immediately. Tx_2 cannot be broadcast and accepted until the *relative* lock_height has passed, relative to the referenced kernel from the earlier Tx_1.

Atomic Swap

This setup can work on Bitcoin, Ethereum and likely other chains. It relies on a time locked contract combined with a check for 2 public keys. On Bitcoin this would be a 2-of-2 multisig, one public key being Alice's, the second being the hash of a preimage that Bob has to reveal. In this setup, we consider public key derivation `x*G` to be the hash function and by Bob revealing `x`, Alice can then produce an adequate signature proving she knows `x` (in addition to her own private key).

Alice has grins and Bob has bitcoin. They would like to swap. We assume Bob created an output on the Bitcoin blockchain that allows spending either by Alice if she learns a hash pre-image `x`, or by Bob after time `Tb`. Alice is ready to send her grins to Bob if he reveals `x`.

First, Alice sends her grins to a multiparty timelock contract with a refund time `Ta < Tb`. To send the 2-of-2 output to Bob and execute the swap, Alice and Bob start as if they were building a normal trustless transaction as specified in section 2.1.

1. Alice picks a random nonce `ks` and her blinding sum `rs` and sends `ks*G` and `rs*G` to Bob.
2. Bob picks a random blinding factor `rr` and a random nonce `kr`. However this time, instead of simply sending `sr = kr + e * rr` with his `rr*G` and `kr*G`, Bob sends `sr' = kr + x + e * rr` as well as `x*G`.
3. Alice can validate that `sr'*G = kr*G + x*G + rr*G`. She can also check that Bob has money locked with `x*G` on the other chain.
4. Alice sends back her `ss = ks + e * xs` as she normally would, now that she can also compute `e = SHA256(M | ks*G + kr*G)`.
5. To complete the signature, Bob computes `sr = kr + e * rr` and the final signature is `(sr + ss, kr*G + ks*G)`.
6. As soon as Bob broadcasts the final transaction to get his new grins, Alice can compute `sr' - sr` to get `x`.

Notes on the Bitcoin setup

Prior to completing the atomic swap, Bob needs to know Alice's public key. Bob would then create an output on the Bitcoin blockchain with a 2-of-2 multisig similar to `alice_pubkey secret_pubkey 2 OP_CHECKMULTISIG`. This should be wrapped in an `OP_IF` so Bob can get his money back after an agreed-upon time and all of this can even be wrapped in a P2SH. Here `secret_pubkey` is `x*G` from the previous

section.

To verify the output, Alice would take `x*G`, recreate the bitcoin script, hash it and check that her hash matches what's in the P2SH (step 2 in previous section). Once she gets `x` (step 6), she can build the 2 signatures necessary to spend the 2-of-2, having both private keys, and get her bitcoin.

Hashed Timelocks (Lightning Network)

TODO relative lock times