Grin Transactions Explained, Step-by-Step

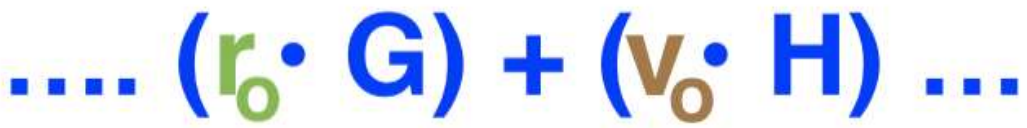
Brandon Arvanaghi Follow
Feb 26 · 12 min read

Grin is an exciting new cryptocurrency leveraging the MimbleWimble protocol. But tutorials on Grin are notoriously nondescript.

This post aims to share **exactly how** Grin transactions work.

. . .

An **output** in Grin is a ***Pedersen Commitment***. Any output will take the following form:



A Grin output, which is a Pedersen Commitment.

A Pedersen Commitment is a clever way to hide information. If this is your first time hearing about commitments, think "shielded value" any time you see that word.

The following, taken from the Grin wiki, is an excellent primer as to what's happening here:

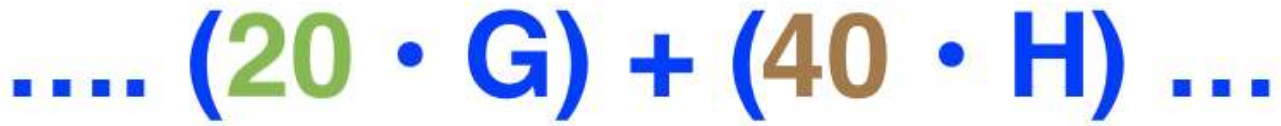
> *If we pick a very large number* k *as a private key,* `k*H` *is considered the corresponding public key. Even if one knows the value of the public key* `k*H`*, deducing* k *is close to impossible...*
>
> • r *is a private key used as a blinding factor,* G *is a fixed point on the elliptic curve and their product* `r*G` *is the public key for* r *on* the curve.

> • v *is the value of an input or output and* H *is another fixed point on the elliptic curve…*
>
> `(k+j)*H = k*H + j*H`*, with* k *and* j *both private keys, demonstrates that a public key obtained from the addition of two private keys (* `(k+j)*H`*) is identical to the addition of the public keys for each of those two private keys (* `k*H + j*H`*).*

A deeper dive into the cryptography can be found in this ECC primer, but in short, to spend a Grin output you must know both the **blinding factor (r)** and the **amount of Grin (v)**. It's impossible to deconstruct a commitment to deduce these values. You must know them ahead of time.

The blinding factor exists because whoever paid you this Grin would also know what **v** is (how much Grin they sent you). But only *you* — not even the sender of the Grin — would know the blinding factor for this output and therefore *only you* are able to spend this output.

Let's say this output uses a blinding factor of 20, and this output consists of 40 Grin. (**Note:** Amounts of Grin are actually sent as multiples of the atomic unit *1 NanoGrin*. I use whole Grin for simplicity here.):



In this output, the blinding factor is 20, and the amount of Grin is 40.

If we look on a Grin blockchain explorer, that output is not broken up neatly like above. Here is what a real Grin output, just like the one we created, looks like:

▼ Outputs (6)

Output Type	Commit	Spent
Transaction	082bd11697071f4b6ef4ba48262cfa118c5b15b6c4f2093a28a8a9f21fe8ec1d4f	False

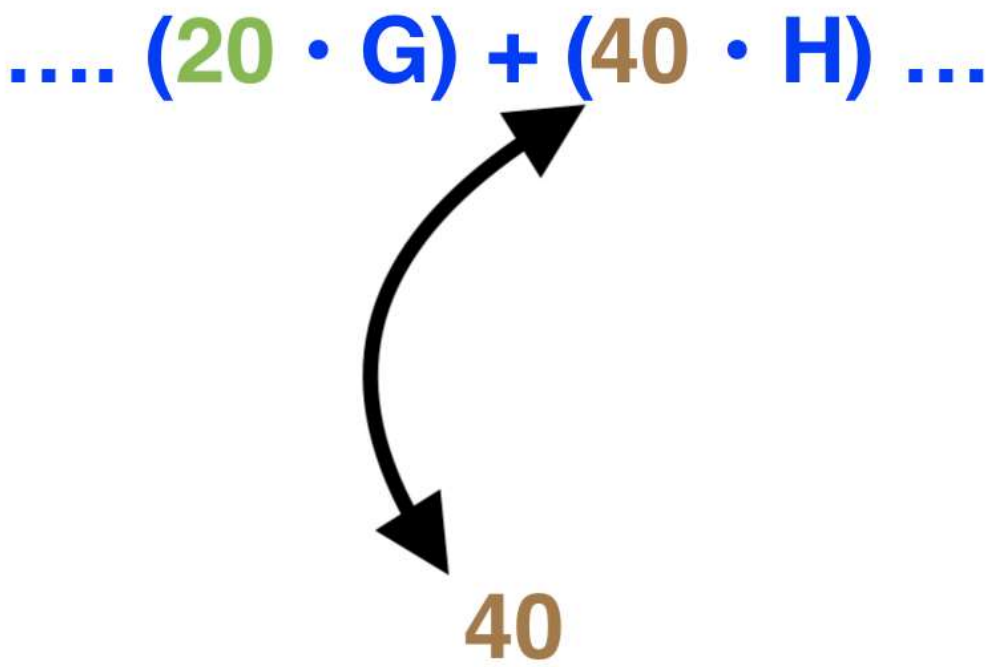
What a Grin output looks like (under the Commit column).

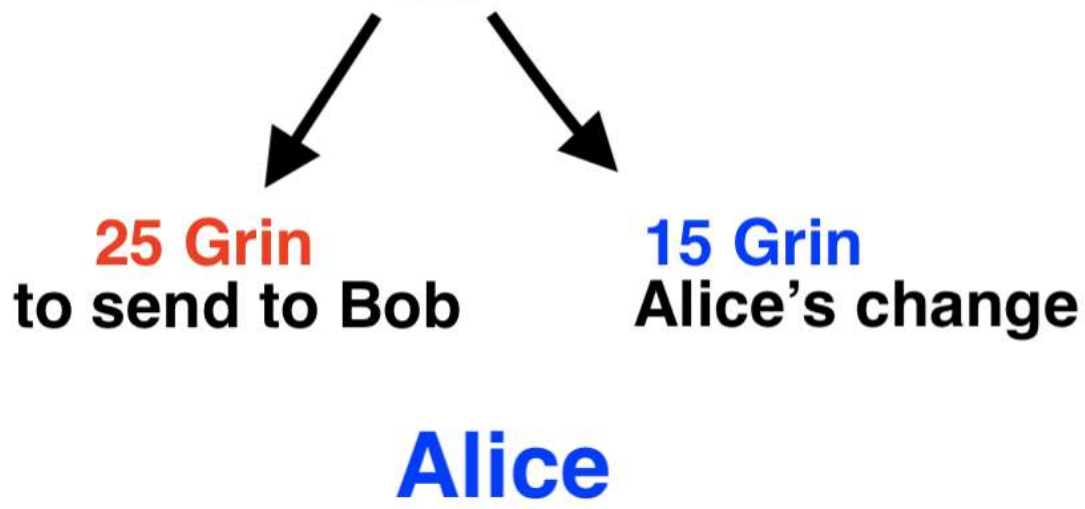
Again, deducing “20” (the blinding factor) or “40” (the amount of Grin) from this output is impossible.

Spending the Output

Say the output we showed you belongs to Alice. Now, Alice wants to send 25 of those 40 Grin to Bob. We'll ignore mining fees for simplicity.

If you have a 5 dollar bill and purchase something for 3 dollars, you get two dollars in change. Bitcoin transactions work like this, and Grin is no different. If Alice wants to send 25 Grin to Bob from her unspent output of 40 Grin, she will also create an output in this same transaction that goes back to herself for the remaining 15 Grin (her *change*).





Alice identifies how much Grin she wants to send to Bob, and also her change.

This 15 Grin will go back to Alice, which means only she should be able to control it and spend it again. In other words, Bob shouldn't be able to spend Alice's change. To that effect, Alice must create a new blinding factor for her change output. Let's say Alice picks **34**.

Alice, knowing both the **r** (the blinding factor for her change output) and the **v** (the amount of Grin, which is her change), has everything she needs to create her change output (**co**). This will be recorded on the blockchain as an output, just like the output Alice will soon generate to send 25 Grin to Bob.

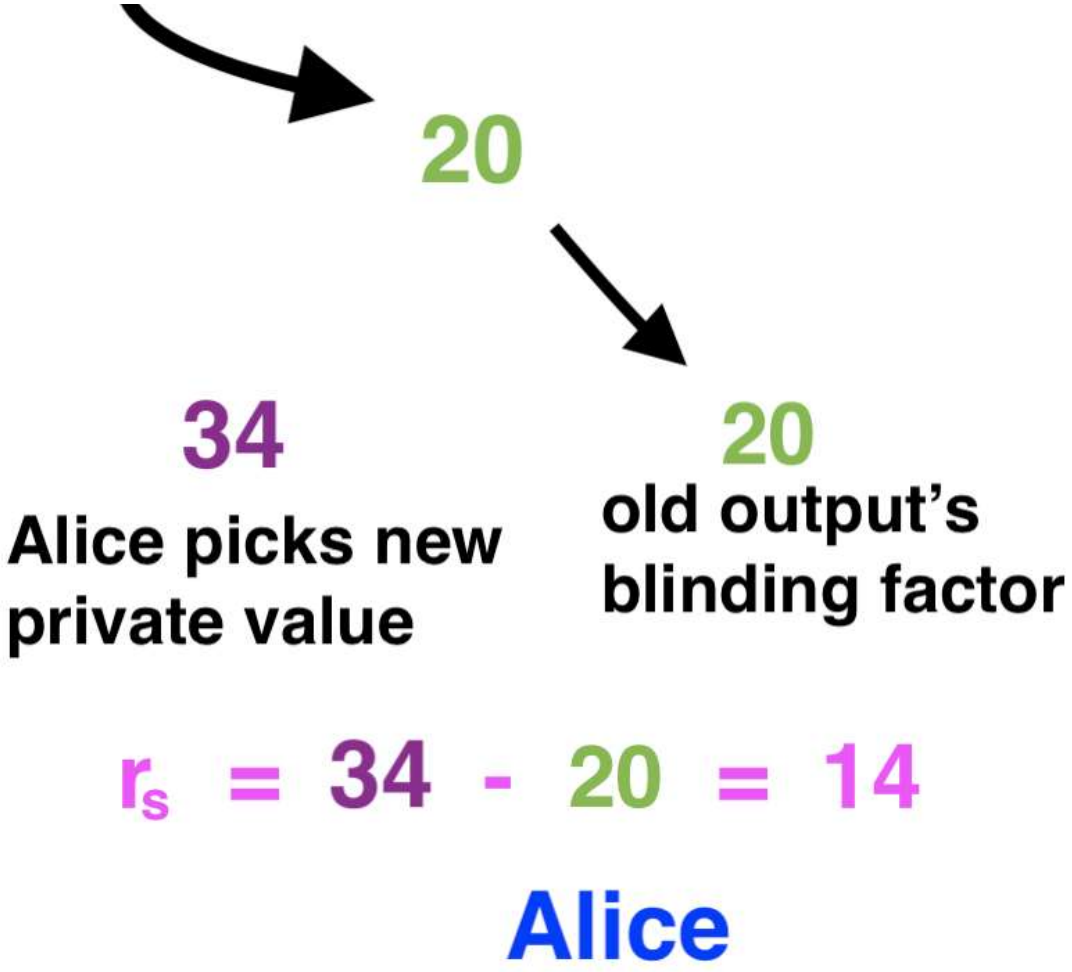
$$CO_{Alice} = (34 \cdot G) + (15 \cdot H)$$

Alice's change output.

As I mentioned earlier, to spend any output, you must know the blinding factor used in that output. Alice knows the blinding factor used in the output she wants to spend (20), but she needs a way to prove to the world that she knows it.

That's why she needs to create an entirely separate calculation known as the *sum of her blinding factors*. This involves Alice taking the blinding factor she just created for her change output (34) and subtracting from it the blinding factor from the output she wishes to spend (20).

$$.... (20 \cdot G) + (40 \cdot H) ...$$

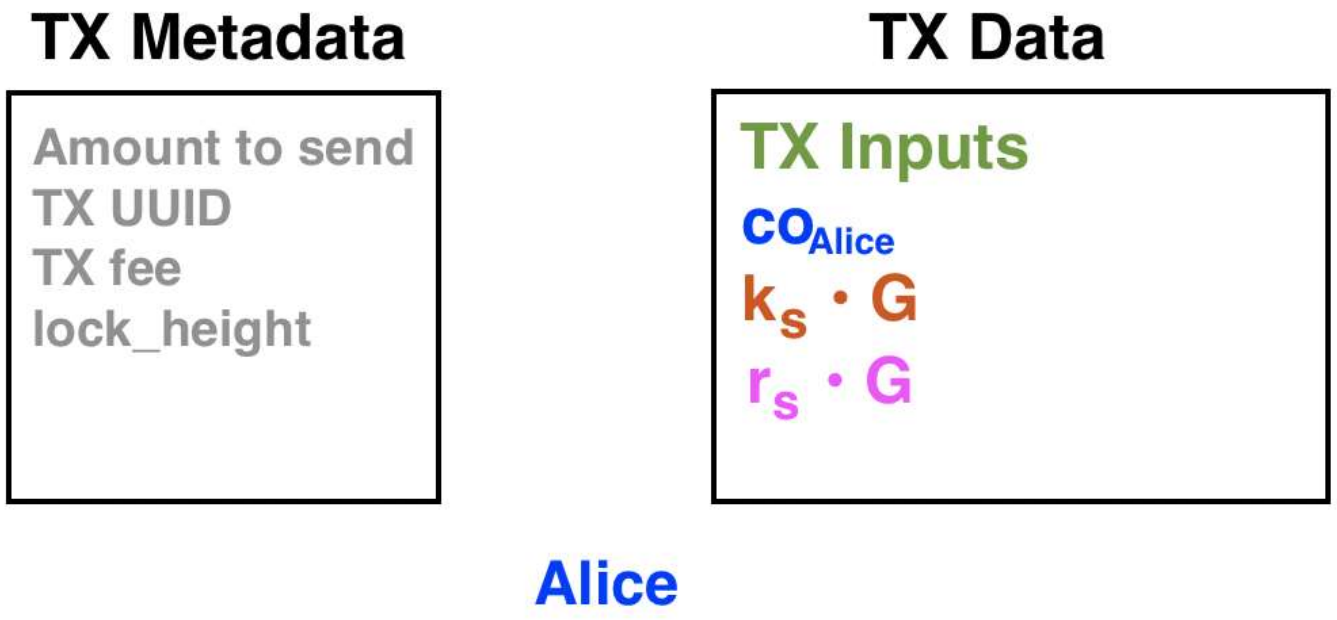


Alice's sum of her blinding factors.

rs (s for *sender*, who is Alice) is known as Alice's *sum of all blinding factors*, and it equals **14** in this case. (**Note:** I am intentionally leaving out kernel offsets).

The last thing Alice does is create a random nonce **ks** (*s*, again, for *sender*). She will use this random nonce to help build her signature for this transaction, which we will show later. Alice doesn't send the actual nonce to Bob. Rather, she sends ***ks • G,*** which is a *commitment* to that nonce. As explained earlier, by multiplying the nonce by generator point ***G***, Alice shields the value of the actual nonce.

Alice sends the following information to Bob. In practice, Grin data are not separated into "Metadata" and "Data" fields, but I do so here for clarity.



Everything Alice sends Bob in the first step of this Grin transaction.

The **Metadata** fields are:

- **Amount to send:** The # of Grin Alice wants to send Bob (in this case, 25).
- **TX UUID**: A unique identifier Alice and Bob use to identify this transaction as they send data back and forth.
- **TX fee:** The transaction fee (which we will not cover in this tutorial).
- **lock_height:** The block number at which this transaction becomes valid.

The **Data** fields are:

- **TX Inputs**: The unspent output(s) Alice uses as inputs for her transaction to Bob.
- **co**: Alice's change output.
- **ks • G**: Alice's nonce **ks** becomes a *commitment* to that nonce when multiplied by generator point ***G***.
- **rs • G**: The sum of all of Alice's blinding factors **rs** becomes a *commitment* to that value when multiplied by generator point ***G***.

Alice sends all of this to Bob, who proceeds with the next step.

Bob's Turn

Upon receiving this data from Alice, Bob concatenates the **TX fee** and **lock_height** variables to create **M**, referred to as the "Message" of the transaction.

The "message" of the transaction.

Bob picks a blinding factor **rr** (r for *recipient*, in this case Bob) for the 25 Grin he expects to receive from Alice. Let's say he chooses **11**. He also picks his own random nonce **kr** (r, again, for *recipient*).

Just like Alice did, Bob creates a *commitment to* both of these values by multiplying them each by generator point ***G***. With these values, Bob

generates the **Schnorr challenge** for the transaction, denoted by the variable **e:**

$$e = \text{SHA256}(M \mid k_s \cdot G + k_r \cdot G \mid r_s \cdot G + r_r \cdot G)$$

The Schnorr challenge for the transaction.

In order, the Schnorr challenge consists of the SHA256 hash of:

- The message of the transaction.
- The sum of the commitment to the nonces used by Alice and Bob.
- The sum of the commitment to Bob's blinding factor (for his output of 25 Grin) and Alice's sum of all her blinding factors.

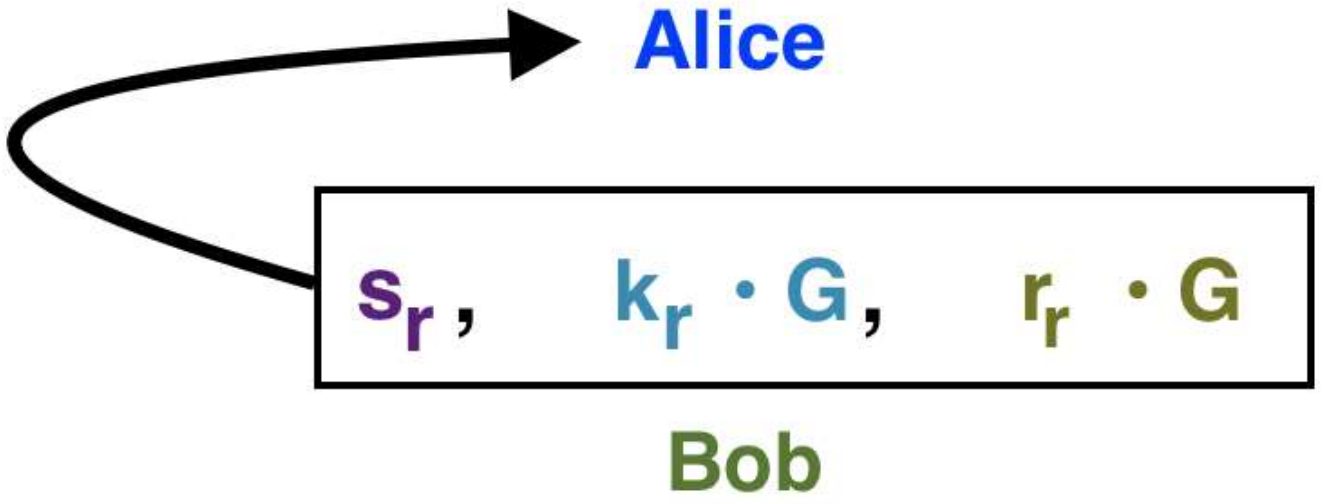
Bob uses **e** to generate his **Schnorr signature** for the transaction, **sr** (*r* for *recipient*). Though it is the entirety of Bob's signature, we call this Bob's *partial* signature, because it will eventually be added to Alice's *partial* signature to create the *signature* of the entire transaction.

$$s_r = k_r + e \cdot r_r$$

Bob's partial signature for the transaction.

When Alice eventually receives **sr**, she will not be able to uncover the values **kr** or **rr** from it.

Bob sends the following back to Alice:



Bob sends his partial signature, commitment to his nonce, and commitment to his blinding factor for his output back to Alice.

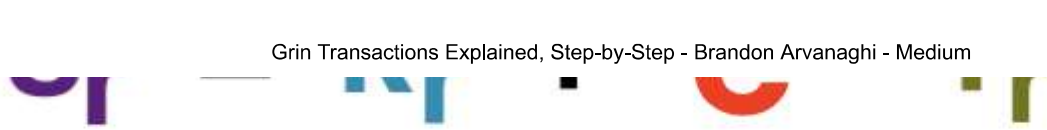
In order, this consists of:

- **sr:** Bob's partial signature.
- **kr • G:** Bob's commitment to his nonce.
- **rr • G:** Bob's commitment to his blinding factor for the 25 Grin he expects to receive.

Last Step: Back to Alice

Alice now has everything she needs to also compute **e**, the Schnorr challenge for this transaction, locally. After having computed **e** locally, Alice can then **validate Bob's partial signature.**

Bob's partial signature **sr**, you'll recall, consisted of the following:



Bob's partial signature for this transaction.

Based on the properties of elliptic curves we described earlier, Alice can introduce the generator point ***G*** into both sides of the equation, and the equality will still hold true.

$$s_r \cdot G = \boxed{k_r \cdot G} + e \cdot \boxed{r_r \cdot G}$$

Alice multiples each side of the equation by generator point G.

Since Alice received **kr • G** (Bob's commitment to his nonce) and **rr • G** (Bob's commitment to the blinding factor he will use for the 25 Grin he expects to receive) from Bob, and since she has already computed **e** locally, Alice can validate Bob's partial signature **sr** by simply multiplying it by generator ***G*** and making sure the right side of the equation equals that value.

In doing so, Alice has proven that:

1. Bob knows how much Grin he will receive (25).
2. Bob knows his nonce.
3. Bob knows his blinding factor for the 25 Grin he expects to receive.

… without Alice ever having known Bob's nonce or the blinding factor he chose.

Alice then generates her own partial signature:

$$s_s = k_s + e \cdot r_s$$

Alice generates her partial signature for the transaction.

Alice can now generate the **signature** of the transaction, which consists of both her and Bob's *partial* signatures:

The signature of the transaction, consisting of the sum of Alice and Bob's partial signatures and the commitment to their nonces.

In order, the *signature* includes:

- The sum of Alice and Bob's partial signatures.
- The sum of Alice and Bob's commitment to their nonces (neither of them know the other's true nonce).

Condensed, this looks like:

$$s = (s, k \cdot G)$$

The signature for the transaction.

where `s = ss + sr` and `k = ks + kr`.

Remember this signature — it will make sense shortly.

Completing the Transaction

Digital money requires a "memory" — that is, when you send money to one person, you can't send that exact same money to someone else. With Grin, we hide both the amount of Grin being sent and the recipients. So how can we prove no money was double-spent or printed out of thin air?

In a Grin transaction, when you subtract all the outputs from the inputs, the amount of leftover Grin should equal 0. Going back to our 5 dollar bill analogy, you have:

```
3 dollars to cashier (output) + 2 dollars in change back to me
(output) - 5 dollar bill (input) = 0
```

In Grin, this same summation makes the *v* values sum to zero when a transaction is legitimate. But how can we prove that without revealing what the values are? Let's look at the inputs and outputs used in our transaction from Alice to Bob:

```
(34•G) + (15•H) + (11•G) + (25•H) - (20•G) - (40•H) = (25•G) + (0•H)
```

The clever property here is that when the Grin amounts cancel out (as they should when no money was created out of thin air), all that remains from subtracting the outputs from the inputs is the commitment to "**the excess blinding factor**", or the "**kernel excess**". This commitment to the excess blinding factor, in our case ***25•G,*** serves as a **public key on the curve**.

If the sum of the outputs of a Grin transaction minus the sum of the inputs produces a valid public key on the curve, you know that the **v** values must have cancelled out. If the right side of the equation is not of the form `n•G + 0•H` for some known value of *n,* you know that the transaction is invalid. That would mean either the amount spent was greater than the sum of the inputs (e.g. you provide a five dollar bill, pay the cashier 3 dollars, and get 10 dollars back in change), or the inputs were greater than the outputs (e.g. you provide a 5 dollar bill, pay the cashier 3 dollars, and there is no change).

Remember the signature from earlier?

$$(s, k \cdot G)$$

The signature from the transaction.

This signature actually **already signed** the **commitment to the excess blinding factor** I just mentioned. Here's how.

If you recall, this is what Bob's partial signature looks like when you multiply both sides of the equation by generator *G*.

$$s_r \cdot G = k_r \cdot G + e \cdot r_r \cdot G$$

Bob's partial signature when you multiply both sides by generator G.

Similarly, here's what Alice's partial signature looks like when you multiply both sides by generator *G*.

Alice's partial signature when you multiply both sides by generator G.

What happens if you add both equations together? You get:

```
sr•G + ss•G = (kr • G) + (ks • G) + (e • (rr•G + rs•G))
```

Remember that **rr** is Bob's blinding factor and **rs** is Alice's *sum of her blinding factors*. Also remember from earlier that `rr•G + rs•G` is the same thing as `(rr + rs)•G` .

Bob's commitment to his blinding factor was *11•G*. Alice's commitment to the sum of her blinding factors was *14•G*. Add them together and you get ***25•G***, which is the commitment to the excess blinding factor for the transaction. Thus, adding **sr** and **ss,** which are Bob and Alice's respective partial signatures, **proves the entire transaction's validity** because they add up to the commitment to the excess blinding factor.

Further simplifying that equation, we get:

```
sr•G + ss•G = (k•G) + (e • (r•G))
```

Or:

```
sr•G + ss•G = (k•G) + (e • (25•G))
```

All that's left to check is that the left side equals the right side.

Remember, everything in this equation (the sum of partial signatures, everything in *e,* the commitment to the excess blinding factor, the commitment to the sum of nonces) is publicly visible to everyone, so anyone can do this verification. We required neither Alice nor Bob's blinding factors to validate the transaction. By adding their partial signatures and verifying that they summed to the commitment to the excess blinding factor, we proved:

1. No money was created from thin air in spending Alice's previous input.
2. Alice and Bob both knew the blinding factors for their outputs when they created this transaction. This means the new outputs are spendable by them, and not lost to the abyss.

The information we just used to validate the transaction gets placed in what's called the **transaction kernel.**

The Transaction Kernel

Besides the outputs, the **transaction kernel** is the other piece of information spit out from a Grin transaction. Every transaction produces a transaction kernel, but there's no way of looking at the Grin blockchain and linking an output to a transaction kernel. One exists for every Grin transaction, and it contains the proof that no money was printed out of thin air.

The following information is stored in the kernel:

- The **signature** of the transaction *(s, k • G)*.
- The public key associated with the "**excess blinding factor**" (in this case, ***25•G***). As described above, this can be used to validate **s**.
- The **transaction fee** and **lock_height** of the transaction. **(Note:** if this was a Coinbase transaction, neither of these would be present).

Summary

After all this is said and done, the only things broadcast to the network from the transaction are:

- The inputs used.
- The new outputs.
- The transaction kernel.
- The kernel offset (which I didn't cover here).

None of the transaction metadata from earlier are relayed. Even better, some of this information may get discarded, too — but we will save that for another post.

Hopefully, this post shed some light on how Grin transactions work. I intentionally left out range proofs, kernel offsets, and fees. Look out for more posts on how cut-through works in Grin, what multi-participant transactions look like, and some experimental features.

. . .

If you enjoyed this post, follow me on Twitter.

See my other post: "What's inside a Grin Transaction File?"

. . .

Thanks to John Tromp, Jasper van der Maarel, and David Burkett for their thoughtful reviews and explainers. Check out their work!

Bitcoin Grin Mimblewimble Blockchain Cryptocurrency