diyhpluswiki/ transcripts/ bitcoin-core-dev-tech/ 2017-09-06-signature-aggregation

https://twitter.com/kanzure/status/907065194463072258

Signature aggregation

Sipa, can you sign and verify ECDSA signatures by hand? No. Over GF(43), maybe. Inverses could take a little bit to compute. Over GF(2).

I think the first thing we should talk about is some definitions. I'd like to start by distinguishing between three things: Key aggregation, signature aggregation, and batch validation. Multi-signature later.

There are three different problems. Key aggregation is where there are a number of people with each their own key, they want to produce a combined key that can only sign when they come together. This is what you want to do for multisig, really. Say 2-of-2 multisig together, the world doesn't need to know that we are 2 of people, so this is a combined address and the money can only be spent when both of us sign. This is done at setup time, before any signature is being made. When you establish your public key, you have done some operations to do this. This is a new key, before anything happens.

Signature aggregation is the problem where you have multiple people who are going to create signatures together but you only want one signature in the end. In this case, there are multiple signatures involved and you just produce a single combined signature. The verifier still knows all the public keys. The output is i this case you have a signature, a pubkey, a message that goes to the verifier. But in the case of signature aggregation, it's a signature, pubkeys, and a message. You need all the pubkeys. This is done at signing time. The signers of this message don't need to do anything ahead of time. Everyone has their key, they don't communicate ahead of time, they create a signature together, they can give it to anyone, with the list of pubkeys, and then they can verify that.

Batch validation is something that is done at verification time where the signers are not involved at all where you have multiple tuples of (message, public key, ... signature). Some of these can be potentially aggregated signatures. You just have multiple things that you are going to verify that are possibly aggregated keys and you're going to verify them all at the same time. You only know whether they are all valid, or not all valid. You don't learn which one is causing a failure. But this is what we care about during block validation. If you fail, you can go back to a scalar validation. Batch validation speedup applies even if not all of them are aggregate signatures. So there's still a performance improvement there.

Much of this work started by looking at key aggregation. There was a nice notion that in Schnorr signatures you can add signatures together and you have a signature valid for the sum of the pubkeys. But then we looked at signature aggregation which is the use case of wanting to have one signature in a transaction. The use case of key aggregation is multisig, the signature aggregation is to reduce the signatures in a transaction to one or a few, and batch validation use case is just performance improvement. Where do the advantages and benefits come from? I am saving that for later in a few moments.

You can do batch validation in a non-interactively aggregated value. Yes. In a moment. There are some more things you can do. I think I first need to explain the formulas a bit there.

The problem of joining all signatures in a transaction into one, we thought we could use key aggregation scheme in the hopes of doing signature aggregation, ... there is a cancellation problem. The naive key aggregation is where you sum keys, with Schnorr signatures you just add your signatures together to sign, which is great, but there's an attack where if you do that protocol then the attack is that the pubkey is the negation of his pubkey plus some other value, and then when you add it up his pubkey gets canceled out and you can sign for it alone. If you use this in signature validation, then it would be vulnerable. The attack is that you see that Greg has an output on the network, you know his pubkey (Pgreg), you create a new output yourself that sends to (Pyours -

Pgreg), and then you create a transaction that spends both of these together, and as (Pyours - Pgreg) + Pgreg sums to Pyours, which you have the private key for, and you can sign claiming I'm both people.

Well, can we find a solution that solves the cancellation problem? Well, we came up with a scheme, then broke it, came up with a better scheme and then a proof, then we wrote it in a paper, submitted it to FC17 and then they said this problem is solved and your proof is not strong enough. They gave some feedback and gave us a paper to look at, Bellare-Neven Multisignatures, which is a variant of Schnorr signatures but natively supports signature aggregation. It doesn't do key aggregation, and can't be used for key aggregation because it only works in the case where the verifier knows all the pubkeys. The scheme we came up with is very similar. We don't have a security proof, and they do.

It's probably best to not do both in the same thing. They are different problems. Key aggregation is not hard. The cosigners need to trust each other. But you can't do that with ECDSA... you can add the keys, but you can't sign without terribleness. There's papers. They use weird crypto and six interaction rounds and craziness.

Is there an attack where you intend to do signature aggregation, one of the keys was done using key aggregation... Yes, that is a risk. But the risk only affects the people who participated in the key aggregation. And that is not a concern because-- if Bryan and I want to do a multisig together, well we need to come up with an address that we both like. If I can claim randomly that X is Bryan and my key, that's already a problem because you are just listening to me. We care about a setup where I can convince Bryan that X is a key for both of us.

Does the multisig thing affect the signature format? Key aggregation doesn't change it. Only solves but is compatible with. Bellare-Neven reduces to Schnorr for a single key and if you do any of the key aggregations, you can use those keys in Bellare-Neven even if it's aggregate. Say there's a multisig and Bryan has the key and I have his key and my own key, and he has verified my key too, then we claim P is the sum of our keys. We tell people that P is our key, people pay to P. And people have also paid to Greg who is just a single key. So there is now Pbryan + Psipa = Pbs. And Pgre. We can create a Bellare-Neven multisig that is verifiable... Someone pays to Pbs and someone has paid to Pgreg and we can jointly construct a single transaction that spends Pgreg and the joined coin with everything you want. The three of us can collaborate to construct this transaction, but the only thing that the world sees is (sig, Pbs, Pgreg, M) where M is the message. And Greg/Pieter there would not be able to figure out a value to trick Bryan, because the signature would be invalid due to the properties of Bellare-Neven multisig. You can sign a message to prove you know the discrete log. The problem of cancellation could exist... Pieter could convince Bryan to not follow the right protocol, and do a cancellation attack. The previous scheme would solve this, but we don't know if it's as secure as Bellare-Neven scheme. The participants can sign with their single key they want to aggregate, to prove they don't have cancelation tricks, but it's interactive. There's a delinearization technique, but without a good security proof. You can pre-sign a message, if you post the signature somewhere, so it does not need to be interactive.

Key aggregation is you among the cosign you sign your public key yourself, which proves you have the private key. This is wallet behavior for multisig. It's compatible with aggregate signature stuff, which is network rules.

andytoshi has implemented Bellare-Neven multisig. It's written, tested, benchmarked, and performance optimizations in progress, but not reviewed yet.

Say x is the private key, P is xG so P is the public key. Then you have k which is the nonce. And then you have R which is nonce times G. And m is the message. Hash function is H. So the Schnorr formula is you have a signature (R, s) which is (k * G, k - H(R || P || M) * x). If you don't include the pubkey in that hash, it's not a proof of knowledge. In bitcoin, the public key is always in the message.

In Bellare-Neven multisig, there's still (R, s) but now it's equal to (k * G, k - H(R || P || C || m) * x1 - H(R || P2 || C || m) * x2), where C = H(x1*G, x2*G). C is just a hash of all these people are participating, and that goes into the hash function as well, you have one term for every ... R is computed with multiple signers. So the way this works is that every participant has their own nonce k1 and k2, and then a corresponding R1=k1*G and R2=k2*G. So you have an interaction round where everyone publishes their own R value, like R1, R2, R3, you add these R values up, and that sum goes into the hash function call. So now everyone can construct this term, you publish all those

terms, so you have someone who creates k1 - H(R || P1 || C || m)*x1, someone creates k2 - H(R || P2 || C || m), and so on, you sum these up, k1 + k2 + k3 becomes k, and those terms are the things that go into the signature. This scheme prevents the cancellation attacks because the hash of every public key is multiplied by is different for each one. The hash changes based on the contents of the pubkeys, so you can't choose a pubkey to cancel out someone else's, because if you do, it changes the hash. If you try to modify your own key to negate someone else's keys, then you change the C, and this is an intuitive argument for why it's impossible. The real proof is in the paper.

What this would boil down to for a Bellare-Neven proposal for Bitcoin, if we want signature aggregation... so we want a thing where there's only one signature per transaction, we add a new opcode that is similar to OP_CHECKSIG but it always succeeds. Instead of actually doing verification, it pushes a message public key pair on to a per-transaction validation context. After all the script checks for transactions are done, you verify the whole thing at once. This is for signature validation, not batch validation. There's a checksig operator, you call it many times, you're passing it into it a message hash and a pubkey, and the message hash is implicit. Every checksig would implicitly compute a message hash, a public key, and signature it detects as input. It doesn't take signature as input anymore, it takes a public key from the stack and it computes a message hash to sign, and just pushes those into a per-transaction validation context. Are the messages different for each signer? Yes. So what's actually signed is the hash of all those signature hashes together with all the pubkeys and you have one designated place in the transaction like say for the first checksig you pass a signature or here's a particular well-defined location, or a witness at the transaction level (but that's incompatible and gratuitous). You can't mix multiple aggregates ? In one of the things, there might be a counter so you could do multiple contexts and then you say which context you're working with, I'm not sure that's necessary, I think there's either all of you use...

The script caching would just-- the scripts don't contain an actual checksig anymore, but you verify them as being .. they automatically succeed. Bitcoin script will never have a conditional checksig. If you have complex smart transactions that might have more ... this only supports the individual things having single aggregated signature rule that is run on everything. So the only coordination between the individual scripts is this one point. What if you wanted a script that wanted to verify a signature from a different transaction. You'd need checksig from stack. .... You could have an OP_IF and wrap it, and then pass on the stack whether you want to run that branch anyway.

You can make your checksigs conditional. There is one group of signers that have to be online at the same time and they can choose to have one signature instead of multiple. But it can be conditional like what that set is, you can have IF branches in your script for that. As long as all the different inputs agree on what the combined shared... yes, that's necessary. Then it should be okay for the individual inputs to pop stuff up to the shared stack and then the verification script that they all agreed on. But the thing you described limits that shared verification that just checks aggregated signatures.. it seems like it might be nice to have more generality functionality? You have a shared stack in the first input it puts it wherever ... OP_CHECKSIGFROMSTACK. Or a special checksig that always checks never batched. Do we need sighash flags? I don't know what the right thing to do. The way this is described, your pkscripts are going to depend on this functionality... you could still use it without the aggregation. You could use the old checksig operator if you want. You could have an input that allows aggregation-compatible checksig, and that input is not made with an aggregate signature, you just provide the signature for it. There might be a reason why you want something Schnorr-like. I started writing a proposal for the bitcoin integration. You have checksigverify operator, takes a pubkey, signature hashes the transaction like normal, then takes a signature. In my draft, you can provide a real signature if it's the first one, or you just provide an aggregate number like 0, and then it shares the signature with someone else, and if not 0 then it's a signature. So you could-- if you had one of these aggregation-spendable outputs, you could spend it without aggregation, it's chosen at signing time as to whether you want to use the aggregation or not. Multiple signers might be online, and the interaction by the way is depending on what security model you want, like 2 or 3 interaction stages between the signers. We could support multiple aggregates in one transaction, it would be straightforward, and the argument to do so would be that because it requires interaction then maybe some inputs are able to aggregate and maybe other inputs are separately able to aggregate. We might try to implement it both ways and see which way is easier. The order of them matters, how do they agree which bucket-- we will probably do-- order matters for signing, because of transaction inputs and outputs, and for performance reasons

maybe the public keys - - eliminate duplicated pubkeys in the aggregation, you don't need to sign twice if the same pubkey is in there. Say 4 inputs in 2 pairs, they need to be in the right 2 pairs, that's where the complexity comes from. To the checksig operator, you provide an accumulator number and every accumulator needs to have one signature and you can have multiple checksigs dependent on it. You don't have to decide that at output time, it's part of the signature, the accumulator numbers go into the signature. One thing you might want to do is put an identifier at the end of the shared stack and say you want to be sure that this UTXO does not get spent with this other UTXO, and this other one will put a unique token on the stack, check that it's not spent with this other one. That's a property of other forms of introspection, it breaks independence of UTXOs. That's a general script feature, not a property of aggregation, with other upgrades to script that's possible so it's not an important part of aggregation. Voting example, you could make it so that you have to have a bunch of things going forever, your shared thing is summing the amounts. You would provide the list of pubkeys, and you identify a subset of them, and then you say for these 51 out of 100 are going to sign, you issue a checksig for those 51, this is independent from aggregation. The script branches for that.

We should explain where the speedup comes from and then explain batch. The reason this is faster is that at verification faster, this formula, this verification equation is one that happens... you gave the signing equation, the verification equation is just multiply the whole thing by G, and then you get s*G = R - H(R || P1 || C || m)*P1 - H(R || P2 || C || m)*P2*. *Yes*. *So I've taken the signature equation, and you get this ... R = H(R || P1 || C || m)*P1 + … + s*G. This is a sum of elliptic curve multiplications. There are efficient algorithms (Strauss, Bos-Coster, Pippenger) that compute these things faster than the individual multiplications and adding them out. The sum of the products is faster to compute than just the individual multiplications. For 10000 keys, the speedup is approximately 8x. This is like over a block. But we're talking about signature aggregation, not batch validation yet. This is a single equation in which there is one R and then a number of terms one for every pubkey and then a single multiplication with G at the end. If you have multiple of these equations to verify, then you can just add these equations up. This is not secure, but it's an initial take-- you have to verify a bunch of these R = blah equations, and then you add them all up, and verify just the sum, as an initial form of batch validation. It works but it's not secure, you can do cancellation where you have invalid signatures and you cancel them out in the sum or something. The solution is to first multiply each of the equations with a random number befreo adding them together. You don't know the ratio between different equations that's the way to protect against cancellation in the other scheme.

This batch validation just basically makes a bigger instance of this multi-exponentiation problem that is fast for lots of inputs. It makes it a bit slower because you are multiplying the R values, so it's a bigger problem. In the end, your entire equation which will contain one multiplication for every key and one for every signature. Take the number of public keys overall, and the number of signatures overall, and there's now a time per point as you go on, so it's completely plausible to get 4000 inputs, because you take all the things since we don't have a .... Pieter is looking up the graph now, I assume. Which he will somehow project on the board.

What percentage of block validation time is signature validation? At initial block download, it's nearly all of it. Assuming you don't have the signatures validated already. It's a majority. Say it's 50%. What's the speedup on verifying the 2 inputs 2 outputs and you assume they are aggregated together, so this is the number 2. So are we talking antminer here? When you're putting stuff in the mempool it's a factor of 1.5x, and when you have an entire block, significantly more. And aggregation shrinks the transaction sizes by the amounts of inputs they have. In our paper, we went and ran numbers on the shrinkage based on existing transaction patterns, there was a 28% reduction based on the existing history. It doesn't take into account key aggregation or how behavior would change on the network as a result of this.

We understand how batch gets a speedup;... and that only applies to full blocks, initial block download, and at the tip we're caching everything and a block shows up-- well, also in the adversarial case. Then there's the thing I like which is where you sum the s values and have G at the end. Instead of using an actual random number generator for the batch validation, you can use deterministic RNG that uses the hash of the block excluding the witnesses or something as your seed, and now the miner can pre-compute these R equations and instead of ... all the s values can get added up, and there should be just a single combined s for the entire block and every individual signature would only have its R pre-multiplied. There are all sorts of challenges for this, such as no

longer validating individually, you would have to validate the whole block. You can't cache the transactions as validation anymore. It's using the hash of the block; you take the merkle root as is, and... so he would have to recompute if he wants to.. It's half the size of all the signatures in addition to what aggregation would otherwise accomplish. It's only aggregating the s values, so you have this signature which is a small part of a transaction and you share half of it, so percentage wise it's not a big change. It doesn't give you any CPU speedup. It saves you from having to do these multiplications with the s values adding them, but that's fast scalar arithmetic, like microseconds for a whole full block.

The more inputs you have, the more you gain because you add a shared cost of paying for one signature, which is a very small advantage but it gives a financial incentive towards coinjoin. Because two transactions will have two signatures, so this incentivizes coinjoin which will have one signature which is cheaper. When you make a transaction, it also increases your incentive to sweep up small UTXOs.

You can do batch validation across multiple blocks, yes. There's this accelerated multi-exponentiation is a sequential algorithm so how do you do that with multiple cores? So if you validate the whole chain in one big batch, you only do one core for that. There possibly can be multi-threaded versions of this if we think about doing millions at once, it may make sense to investigate. We've been looking into efficient algorithms for multi-exponentiation. Andrew implemented one, Jonas Nick implemented one, it's a novel combination of parts, and perhaps we should write about it.

The FC17 paper was not published (it was rejected) because of the Bellare-neven multisig work, so we'll put together something else, or just ask us for the previous paper.

These multiplication algorithms will probably go into libsecp256k1 even without aggregation. And then writing up a BIP. We simplified this process because before we had this custom delinearization scheme... we had this extra chunk where we had to go get public review from cryptographers. Bellare-Neven is widely-reviewed and well known. We looked for this for a while, but only now people are suggesting it. We found lots of papers that ignored the cancellation problem. They call it "multi-signature". The title of the paper makes it hard to find: “Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma”. Their most important contribution was proving that this is secure (and their strategy for this).

Get the crypto finished and tested in libsecp256k1, and concurrently with that, we create a BIP for bitcoin that will implement a new script version, which has a new checksig operator and maybe some other things will ride together on that. So we redefine checksigs to be this, or maybe this is the only upgrade in this script version. It seems like the different signature type is a more basic thing easier to understand, and aggregation is another layer on top of this. Slightly smaller signatures, and we get schnorr aggregation, but you can do aggregation in ECDSA but it's pretty complex to implement and basically nobody has implemented. And this other stuff is more like a couple hundred lines of code. Aggregation for schnorr is pretty easy. If you introduce a new signature type, you can do batch validation, but you can also do batch validation with ECDSA by including 1 extra bit with every signature. I'm worried about it looking like a .... The biggest hurdle here is that it crosses a layer, we're introducing a transaction-level script checking thing, well checklocktimeverify which is a transaction-level.... The signature aggregation across inputs; checklocktimeverify is something you verify inside the script that looks at the transaction. But this other stuff is more like locktime more than checklocktimeverify. You still need to look at the signatures. This is a new level of validation that gets introduced. I think it's worth it, but it's somewhat gnarly. You have this new validation context where you can introspect things that you couldn't do before.. No, no introspection. It's just aggregation. You are merging signatures together. That's all. You should not add other riders to this kind of upgrade; if you want to do that, introduce another script version that does not touch the cryptography.

In every output, make every input look indistinguishable. That is one of my goals. Nobody in the network should be able to see what your script is. This is a step towards this. Validation time regardless of how complex your script is. You just prove that the script existed, it is valid, and the hash. But anything that further complicates the structure of transaction. This helps fungibility, privacy, and bandwidth. Having a transaction-level conditions across inputs.... checking the aggregation fee for your transaction. Sum of the inputs have to be greater than the sum of the outputs. This is similar to scriptless script reasons.

Simultaneously with one unit of work, compute a reciprocal square root at the same time as the inverse. From one of the Bouncycastle developers. He used this to come up with a way to do an X-only for libsecp(?)... and you get only the X out. With the cost of an extra decompression, ... the right X. You have like an X and you have your scalar, and you take the X and there's an isomorphism to a jacobian point on an isomorphic curve. You have now converted it into an isomorphic curve with coordinates x, x squared, x cubed or something like this, and then you can multiply that by ... ... and you get this point at the end which is in projective coordinates.. and so in order to get the correct value using that projection from an isomorphic curve to the correct curve, you can do a square root, you can move the square root from the beginning to the end. You have to do an inverse to get out of the coordinates. It's super fast ECDH for this curve, and that's the big driver for co-factor 2 curves, ... you need to do montgomery multiplier which only works on curves with a certain cofactor. Maybe put it into BIP151. You save a byte, so....

All the 32-byte things are reversed, and all the 20-byte things are the right way. The RPC returns it the other way. RPC writes 20-byte things the right way. Bitcoin displays them all backwards. Bitcoin interprets 32 byte sequences as bigint little-endian format, printed to the users as big-endian. Everything is little-endian.

---

https://blockstream.com/2018/01/23/musig-key-aggregation-schnorr-signatures.html

https://eprint.iacr.org/2018/068

Last edited Thu Jan 25 12:03:37 2018