Bitcoin Technology Development Talks: Schnorr signatures

2018-02-01

Bryan Bishop
kanzure@gmail.com
https://twitter.com/kanzure

0E4C A12B E16B E691 56F5 40C9 984F 10CC 7716 9FD2

This document contains transcripts from various bitcoin development talks, mostly about Schnorr signatures and signature aggregation. Any errors are most likely my own.

Schnorr signatures for Bitcoin (2018)

http://diyhpl.us/wiki/transcripts/blockchain-protocol-analysis-security-engineering/2018/schnorr-signatures-for-bitcoin-challenges-opportunities/

Pieter Wuille (sipa)

2018-01-31

video: https://www.youtube.com/watch?v=oTsjMz3DaLs

slides: https://prezi.com/bihvorormznd/schnorr-signatures-for-bitcoin/

https://twitter.com/kanzure/status/958776403977220098

Introduction

My name is Pieter Wuille. I work at Blockstream. I contribute to Bitcoin Core and bitcoin research in general. I work on various proposals for the bitcoin system for some time now. Today I will be talking about Schnorr signatures for bitcoin, which is a project we've been working on, on-and-off, for a couple of years now. I'll talk about the cool things that we can do that might be non-obvious, and also some non-obvious challenges that we ran into while doing this. This is, as this talk covers things we've done over a long time, there are many other people who have contributed to this work, including Greg Maxwell, Andrew Poelstra, myself, and also Russell O'Connor and some external contributors including Peter Detman and others. I wanted to mention them. And Jonas Nick.

Benefits of Schnorr signatures for Bitcoin

Schnorr signatures have been talked about for a while. The usual mentioned advantages of this approach are that we can decrease the on-chain size of transactions in bitcoin. We can speed up validation and reduce the computational costs. There are privacy improvements that can be made. I'll be talking about those and the problems we've encountered.

Bitcoin

For starters, let's begin by talking about bitcoin itself. Transactions consist of inputs and outputs. The outputs provide conditions for spending. Russell was talking about this in his previous talk. They are effectively predicates that need to be satisfied. Inputs provide the arguments to those predicates. Typically, in the outputs, the predicated that is included is required signature with key x. This is the most common, but it's by no means the only thing that we can do.

Bitcoin also supports threshold signatures in a very naieve way. Threshold signatures are schemes where you have a group of n possible keys and you decide ahead of time that any subset k out of those n are able to provide a valid signature and with less it's not possible. Bitcoin does this naively by giving a list of all the keys and all the signatures. It's an obvious, naieve way of implementing this construction but it's by no means the best that we can do.

Predicates and signature validation

Important for what I'll be talking about later is that in the blockchain model, the chain itself, meaning all the full nodes that validate the chain, do not actually care who signs. If there are multiple possible signers, for example I have wallet on my desktop computer but I want to make sure that I'm protected against software attacks maybe I also want a hardware device and I want to use a system where both wallets need to sign off on a transaction. This is 2-of-2 or 2-of-3 multisig construction. If I want someone to pay me, I am the one who is going to decide what conditions they should be creating when sending the money. I'll tell them "create an output of this much money to an address that encodes 2-of-3 multisig and these are the keys for that" and we have a compact 2-of-3 pay-to-scripthash (P2SH) implementation for that. But it is me who cares who those signers are. It's not the blockchain. The chain only sees that yep there was supposed to be a key with this signature, and then it simply sees and validates the presence of this.

Scripts

Bitcoin accomplishes this through scripting. It's a scripting language called Bitcoin script. It's a stack-based machine language. The most simple example you can come up with is an output that says "pubkey CHECKSIG" and then an input that contains a signature. The execution model is that you first execute the input, which results in a signature on the stack. Next, you execute the output commands which pushes the public key on to the stack and the CHECKSIG looks at both the signature and the pubkey and checks whether the transaction is good to go.

In practice, what happens is that I don't tell my senders an actual public key. Instead, I give them a hash of my public key. This was originally for compactness reasons but there are other advantages. You can see the script that is being used for that. Effectively what the script does it takes two inputs, a signature, and a public key. Verify that the hash of the public key is x, and that the signature is valid for that public key.

Going forward, we will be talking about threshold signatures. Bitcoin's way of dealing with threshold signatures is through an opcode called OP_CHECKMULTISIG which takes a number of keys and a number of signatures, matches them all, and here you can see how this works.

Other things that bitcoin scripting language can do includes hash preimages and timelocks which are used in various higher-level protocols. I should also say that bitcoin script uses ECDSA. It's a common standard. But let's talk about Schnorr signatures.

Schnorr signatures

Schnorr signatures are a well-known signature scheme that only relies on the discrete logarithm assumption just like ECDSA does. There are various advantages for Schnorr signatures over ECDSA. Some of them are that it supports native multisig, where multiple parties jointly produce a single signature. This is very nice because we can reduce the number of keys and number of signatures that need to go into the chain. There are various schemes that enable threshold signatures on top of Schnorr signatures. In fact, it is known that there are constructions to effectively implement any monotone boolean function. I'll talk a bit about that.

Monotone boolean functions are the class of functions from booleans to booleans that can be written using only AND and OR gates. As long as we restrict ourselves to spending conditions that consist of some group of people signing or some other group signing, then this is exactly the class that we want to model. It is in fact known that there are schemes that might have complex setup protocols but it's actually possible to negotiate keys ahead of time in such a way that A and B or B and C and D, or D and F, or whatever, can eventually sign for this key.

We recently published a paper about a scheme called MuSig which does native multisignatures but without any setup. I'll talk a bit more about this later.

Other advantages of Schnorr signatures is that they support batch validation where you have multiple sets of keys and messages and you can verify them all at once in a way that is more computationally efficient than doing single validation.

Schnorr signatures have a security proof, which is not true for ECDSA. In addition, they are also non-malleable, so a third-party that does not have access to a private key cannot modify the signature without invalidating it.

Simply by virtue of introducing a new signature scheme, we could get a number of advantages for free. One of them is that ECDSA signatures in bitcoin right now use the rn encoding which adds 6 bytes of completely unnecessary data to the chain for every signature. We can just get rid of this once we introduce another signature scheme.

Most of the things on this slide are in theory also possible with ECDSA but it comes with really complex multi-party computation protocols. I'll talk about in a minute where this is not the case.

Can we add this to Bitcoin script?

Seems like an almost obvious question and win. We can make the same security assumptions. There are only advantages. Same security assumptions. Ignoring the politics for a second, it seems like we could add a new opcode to the scripting language which is especially easy since bitcoin now has segwit activated and part of that system added script versioning which means we can introduce new or proposed new scripts with new semantics from scratch without much effort. There are other advantages that come from this, though. I'll talk about two of them.

Taproot

One scheme that can benefit from this sort of new Schnorr signature validation opcode is taproot, which if you've been following the bitcoin-dev mailing list over the past few days you may have seen mentioned. Taproot is a proposal by Greg Maxwell where effectively the realization is that almost all cases where a script gets satisfied (where an actual spend occurs) and there are multiple parties involved can almost always be written as "either everyone involved agrees, or some more complex conditions are satisfied". Taproot encodes a public key or the hash of a script inside just one public key that goes on to the chain. You cannot tell from the key whether it's just a key or if it's a key that also commits to a script. The proposed semantics for this allow you to either just spend it by providing a signature with the key that is there, or you reveal that it's a commitment to a script and then you give the inputs to satisfy that script. If that signature used in the taproot proposal was a Schnorr signature, then we get all the advantages I talked about for Schnorr signatures. So not only could this be used for a single signer, but it could also be the "everyone agrees automatically" by using a native Schnorr multi-signature.

Scriptless scripts

Another area that Schnorr signatures can help with is the topic of scriptless scripts, an area that Andrew Poelstra has been working on. There was a talk about this recently at RealWorldCrypto 2018 which I think was very good. The idea here is how much of the features of an actual scripting language can we accomplish without having a scripting language? It turns out, quite a lot. In particular there is a construction called a cross-chain atomic swap which I won't go into the details here but it allows multiple-- so, I want to sell someone some bitcoin and someone else wants to sell me some litecoin and I don't know why but assume it's the case and we want to do this in lockstep across the chains so that no party is fraudulent. Both transactions have to be reversible, so that the other party can't back out. A cool construction for this was proposed a couple of years ago. It's a cross-chain atomic swap where the second payment is dependent on using a hash preimage which gets revealed by the other transaction. We put the coins into a construction where they are locked and then when one party takes out their part of the coins, they reveal the information that I need in order to take their other coins. This makes the whole construction atomic. The normal formulation of this always requires on the hash preimage and revealing that and so on. But it's possible with just a Schnorr signature and this makes it indistinguishable from a normal payment and it also makes it smaller. Schnorr signatures will fit in well with the scriptless scripts scheme and cross-chain atomic swaps. There are many things we can do with Schnorr signatures. We want this.

Cross-input validation

Why stop at one signature per input? Bitcoin transactions have multiple independent outputs and we don't want to restrict the ability for someone to choose them independently. All of these have public keys and signatures that are required. Why can't we combine all of those signatures into one? Schnorr signatures support multisig so this seems like an obvious win. Well, not so fast.

I'll show a few equations. The message is m. The public key is X where X = x * G where x is the private key and G is the generator. The signature is a tuple (R, s) which is valid if s * G is equal to R + Hash(X, R, m) * X. What you can notice about this equation is that it is linear. All of the public keys really appear at the top-level which means that what you can do is if multiple parties effectively produce an s independently for the same R value or for some of the R values and you add up the s values the result is a signature that is valid for the sum of their keys. This is how all multisignature constructions for Schnorr work. They are all based on this principle.

Unfortunately there is a caveat here, called the rogue key attack. Assume Alice has key A and Bob has key B. Bob claims that his key is B prime which is really B minus A. So Bob claims that's his key and people believe him. A naieve multisignature would use the sum of the keys and (B', A) is really just B which Bob could sign for without Alice's cooperation. Everyone see's Alice key but Bob's says I send to the sum of these keys and I assume that this will only be spendable by both Alice and Bob and this is wrong. The normal way to prevent this is to require the keys to sign themselves. This is effectively an enrollemnt procedure or certification procedure or you include with the public keys a signature that signs itself. There are various constructions but you must guarantee that the parties actually have the private keys corresponding to the public key that they claim to have.

This works for multisig within a single input approach because the people who care about it are just the participants and they can internally prove to each other yep here's my key and here's a proof that it's my key and it doesn't go into the blockchain. But for cross-input aggregation, we want to reduce all of the inputs in the transaction to one, this is actually not possible, because the sets of keys are under control of the attacker. So the example again is that Alice has a number of coins associated in an output with key A, and Bob wants to steal them. We use a naieve multisig approach with a signature with the sum of all the keys that we can see. And Bob can create an output in a transaction himself of some marginally small amount addressed to the key B minus A and then create a follow-up transaction that spends both Alice's coin and Bob's coin in such a way that they cancel out. So this is a completely insecure situation and I believe the only way to prevent it is by including the self-certification signature inside of the blockchain itself, which would undo all the scaling and performance advantages we assumed to have.

What we need is security in the plain public key model where there is no key setup procedure beyond just users claiming that they have a particular key. They are allowed to lie about what their key is, and the system should remain secure. This was something we noticed and we tried to come up with a solution for this rogue-key attack. We tried to publish about it, got rejected, and we were told that we should look at a paper from Bellare-Neven 2006 which exactly solved this problem.

Bellare-Neven signatures

The Schnorr multisignature is S * G = R + H(X, R, m) * X where X was the sum of the public keys. Bellare-Neven introduced a multisignature where you use a separate hash for every signer. Into every hash, goes the set of all the signers. The great thing about this paper is that it gives a very wide security proof where the attacker is allowed to pretty much do anything. An attacker can participate in multiple signing attempts with multiple people simultaneously. This looks exactly like the security model that we want. So let's go for this and start thinking about how to integrate this into Bitcoin script.

Again, not so fast. There's another hurdle. We need to consider the distinction between a multisignature and an interactive aggregate signature. The distinction is that a multisig is where you have multiple signers that all sign the same message. In an interactive aggregate signature, every signer has their own message. Seems simple. In the context of bitcoin, every input is signing its own message that authorizes or specifies the claim authorizing the spend. There is a very simple conversion suggested by Bellare-Neven themselves in their paper where you can turn the multisignature scheme into an interactive aggregate signature scheme where you just concatenate the messages of all the participants. This seems so obviously correct that we didn't really think about it until my colleague Russell O'Connor pointed something out.

Russell's attack

https://www.youtube.com/watch?v=oTsjMz3DaLs&t=22m27s

Russell pointed out that we've-- let's assume that Alice has two outputs, o1 and o2. Bob has an output o3. And we assume m1 is the message that authorizes a spend of o1, and m2 the same for o2, and so on. Alice wants to spend o1 in a coinjoin with Bob. So there's a multi-party protocol going on, mentioned in an earlier talk that coinjoin is where participants get together and move their coins at the same time and now you can't tell which outputs belonged to which inputs. It's a reasonable thing that Alice and Bob would want to do this. In this protocol, Bob would be able to claim he has the same key as Alice. It's perfectly allowed in the plain public key model. And he chooses as a message, m2, the message that authorizes the spend of Alice's second output instead of m3 his own output. And you may claim well it's perfectly possible to modify the protocol where you say don't ever sign something where someone else is claiming to have your keys. But this is a higher-level construction that we would like the underlying protocol to protect against this sort of situation. If you now look ta the validation equation becomes, you see that Alice's public key appears twice, and the concatenation of the two messages appears twice, but these two hashes are identical. So Bob can duplicate all of Alice's messages in a multi-party protocol and end up with a signature that actually authorizes the spend of Alice's second output which was unintended.

Mitigating Russell's attack

A better solution that we are proposing is that instead of this L which is the hash of the commitment of all the participant public keys in the set, you include the messages themselves and then in the top-level hashes you include your position within that hash. Russell's attack doesn't work anymore because the messages in every hash are different so Bob can't just repeat the message and steal things. So something to learn about this, at least for myself, is that attack models in multi-party schemes can be very subtle. This was not at all an obvious construction.

Bitcoin integration

Here I guess I should do the slide that I had before. Sorry if I'm making you sea sick. Concretely, how do we integrate this Bellare-Neven like interactive aggregate signature scheme into bitcoin? It seems to give us a lot of advantages. We can turn all of the signatures in one input into a single signature using multisig and threshold signatures. And we can use cross-input aggregation across multiple inputs to even reduce that further and only have one signature for the entire transaction.

How do we do this? There's a hurdle here. Bitcoin transactions are independent. We have this model where there is an output with a predicate, you provide an input with all the arguments needed to satisfy it and the transaction is valid if all of the predicates are satisfied plus a number of other constraints like "you're not double spending" and "you're not creating money out of nothing" and all those things.

For cross-input aggregation, we want one signature overall. The way to do it, or at least what I would propose, is to have the CHECKSIG operator and the related operators always succeed. Right now they take a public key and a signature from the stack and validate whether they correspond. Instead, make them always succeed, remember the public key and message, compute what the message woud have been signed. Continue with validation for all the inputs in a transaction. Now tha tthe transaction is validated, and if all the input predicates succeed still, but in addition there is an overall Bellare-Neven interactive aggregate signature provided in the transaction that is valid for all the delayed checks. This is a bit of I guess a layer violation but I believe it's one that is valuable because we get all these savings.

Performance

I want to talk a bit about the actual work we've been doing towards that end. I want performance. Andrew Poelstra, Jonas Nick and myself have been looking at various algorithms for doing the scalar multiplication in the Bellare-Neven verification equation and there are various algorithms that you get better than constant speedup. You can compute the total faster in aggregate or batch than computing the multiplication operations separately and adding them up. This is a well-known result, but there's a variety of algorithms. We experimented with multiple of them. In this graph you can see how many keys were involved in the whole transaction, and then the speedup you get over just validatin those keys independently. You have two alorithms- one is Strauss and the other is Pippenger. After various benchmarks and tweaking at what the correct point is to switch over from one to another. Initially for small numbers, Strauss algorithm is significantly faster but at some point Pippenger gets faster and it realy goes up logarithmically in the number of keys. This seems to continue for quite a while. Our overall validation speeds for n keys is really n over log n if we're talking about large numbers.

You may think, well, there's never going to be a transaction with 10,000 keys in it, right? You're already doing cool threshold scheme so there's only one key left so you don't need to think about the extreme cases. But this is where batch validation comes in because Bellare-Neven's validation equation can also be batch validated where you have multiple instances of the equation that can be validated in parallel and you only need to care if they all fail not which specific one fails because that's the block validity requirement in bitcoin. You're seeing multiple transactions in a block and all that you care about is whether the block is valid.

These performance numbers apply to all the public keys and signatures you see in a transaction, within a block, rather than just within a transaction. And within a block, we potentially see several thousands of keys and signatures, so this is a nice speedup to have.

Space savings

Furthermore, there are also space savings. This chart is from a simulation where we assume that if this proposal would have been active since day 1 then how much smaller would the blockchain be. Note that this does not do anything with threshold signatures or multisig and it doesn't try to incorporate how people would have differently used the system (which is where the advantages really are) but this is purely from being left with one signature per transaction and everything else is left in place. You can see between a 25% and 30% reduction in blockchain size. This is mostly a storage improvement and a bandwidth improvement in this context. It's nice.

Ongoing work

We're working on a BIP for Bellare-Neven based interactive aggregate signatures. We can present this as a signature scheme on its own. There's a separate BIP for incorporating this cross-input capable CHECKSIG and its exact semantics would be-- I lost a word here, but the recommended approaches for doing various kinds of threshold signings so that we don't need to stick with this "everyone involved in a contract needs to be independently providing a signature" scheme.

That's all.

Q&A

https://www.youtube.com/watch?v=oTsjMz3DaLs&t=33m3s

Dan Boneh: We have time for questions. Any hope of aggregating signatures across transactions? Leading question.

A: I expected something like that from you. So, there is a proposal by Tadge Dryja where you can effectively combine even across transactions- you can do a batch validation ahead of time and look at what multipliers you would apply and on the R value you can combine the whole R value into a single one. However, this is even more of a layer violation in that transaction validation comes with extra complications like what if you have a transaction that has been validated ahead of time and its validation was cached but now you see it in inside of a block and you need to minus-- what you're aiming for is less signatures where you can arbitrarily and non-interactively combine all signatures. I think that's something we should look into, but I would rather use all of the possibilities with the current existing security assumptions and then perhaps at some point later consider what could be done.

Q: Hi. I was wondering one question about taproot. The introduction of this standard case where everyone signs and agrees would basically reduce the number of times where you see the contract being executed at all. Wouldn't this reduce the anonymity set?

A: I don't think so because in the alternative case where those people would have a contract that explicitly stated everyone agrees or more complex setup-- you would still see, you're going from three cases. One is just single signature single key, to everyone signs with multiple keys and third more complex constructions. What we're doing with taproot is unifying the first and second branch but the third isn't effected. I think this is strictly not the case.

Q: You had alluded to political reasons why this wouldn't get merged. What are the reasons against this?

A: I would very much like to see what I've been talking about today to be merged into bitcoin. It's going to be a lengthy process. There's a long review cycle. This is one of the reasons why I prefer to stick with proposals that don't change the security assumptions at all. None of what I've been talking about introduces any new assumptions that ECDSA doesn't already have. So this hopefully makes it relatively easy to see that there are little downsides to deploying this kind of upgrade.

gmaxwell: An extra elaboration on the taproot point... you're not limited to have the "all agree" case. It can be two-of-three. If your policy was 2-of-3 or 1-of-3 with a timelock then the case that looks like just a single key could just be the 2-of-3 at the top. This would be another factor that could help the anonymity set situation.

Q: Does the Schnorr signature.. I'm wondering about its availability for open-source or widely available software like openssl? My business case would be just.. update.. signature, done by multiple parties.

A: The most commonly deployed Schnorr-like signature is ed25519 which is very well known and used in a number of cases. I believe there are higher-level protocols that specify how to do aggregating multiple keys together and sign for them at once. You may want to look into a system called cosi.

"Design approaches for cross-input signature aggregation"

Schnorr-based signatures in Bitcoin (2017)

http://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/schnorr-signatures/

Pieter Wuille (sipa)

https://twitter.com/kanzure/status/785040458103791616

http://pieterwuillefacts.com/

slides: https://prezi.com/09lr4goiujol/schnorr-signatures-in-bitcoin

https://youtu.be/_Z0ID-0DOnc?t=2297

I don't consider myself to be a cryptographer. Most of the work I will be talking about is mostly the result of talking with a lot of smart people. These are things we have been talking about for a long time. Many issues have come up, and I'm glad that it has taken a while. This is the first time I have seen my slides, my screen was messed up when I spilled coffee and all the colors were messed up - it was really interesting but I don't understand it.

Schnorr signatures for Bitcoin, I will first talk about Schnorr signatures and then about bitcoin.

Schnorr signatures

They are a cryptographic system. That's the formula. I don't think I will discuss the details here. I will first talk about history of how we got to the situation we are today with ECDSA in bitcoin and then talk about the advantages that Schnorr signatures could and would have, and how to standardize that, and then go through applications that they could have and then show that the problem is harder than swapping one for the other.

So history... Schnorr signatures were originally proposed in 1988 by Schnorr who patented his invention. The nice thing about Schnorr signatures is that they are remarkably simple. They are easier than ECDSA, even. They work in any group where the discrete logarithm is hard. At the time they were proposed for multipliation of modular groups. However, in 1993, a standard for signatures based on this type of cryptography was standardized, however they didn't use the Schnorr system presumably because it was patented. Instead, DSA was standardized, presumably to avoid some of the patents. Schnorr claimed for a long time that DSA infringed on his patents.

In 2005, people built on top of DSA rather than Schnorr signatures. They could equally be applied to elliptic curves. Bitcoin in 2008 used ECDSA because that was the only standardized elliptic curve. In 2011, Ed25519 was standardized by djb which is essentially a very close group. Schnorr signatures are not an established standard. ECDSA is documented in ways that exactly specify all the math that have to happen. How to serialize the signatures, how to serialize the public keys and what each bit exactly means. This is not the case for Schnorr, which is more of a general idea for how to build a system. Well what I am going to try to do is convince you that we need a Schnorr signature standard.

Under standard assumptions, like the random oracle model and the assumption that the discrete logarithm problem is hard, ECDSA is secure. Schnorr is provably non-malleable. With low-s policy and segwit, this prevents known malleability of ECDSA but there's no proof of no other malleability in ECDSA, but this is not an issue in Schnorr signatures. You can verify in batches for Schnorr signatures at high speed in each of them individually. This is exactly what we want for bitcoin blocks because they are essentially big batches of signatures to validate. Also, in Schnorr, you can have native k-of-k multisignatures, you can get a bunch of keys together and have a single signature that proves that all of them sum.

Applications of Schnorr signatures

Can we take Schnorr as a drop-in replacement for ECDSA in bitcoin? And can we apply it to multisig transactions? And then I will talk about signature aggregation. My goal here is to come up with a single standard that fits all of the applications so that we don't have to worry about what can be used where and when.

So first, drop-in replacement question. The security proof of Schnorr signatures says that they are existentially unforgeable under the assumptions I mentioned before. What this means is that there is a fixed chosen public key in advance that it is impossible to create a message for that key for any message even those that an attacker can choose. Turns out this is not precisely what we want. It doesn't say anything about keys that you haven't chosen in advance. If you take Schnorr sigantures and apply it to an elliptic curve group, it has a really annoying interaction-- if you know a master public key and you see any signature below it, you can transmute that signature into a valid key for any other key under that master key. This is not necessarily a problem under standard assumptions but what I'm trying to tell you is that we should really test our assumptions. This nice proof of existentially unforgeable is nice, but we need to test whether that's the only thing we want.

Thankfully there is a known solution for this, it's a recommendation that inside Schnorr signatures there is a hash function in the formula, you have the public key under the hash. Or in other words, the message you are signing is not just the message, but it's the concatenation of the public key and the message and the problem disappears. The problem with this is that this allows naieve key recovery. If I give you a signature and a message, you could have derived the public key that would have signed this. We don't actually need that property.

Multisigning is the big advantage for Schnorr and the whole reason we want this. A group of people can jointly create a signature that is valid for the sum of their keys. In the slide here, you see U1 through U3 which are users and how this mechanism works is that there is a two-round interaction scheme where they each pick nonces, they communicate to each other and add them together to get an overall r value which everyone knows, and then signs using this nonce with their own key getting three signatures, then you combine all the s values into the final s value which is a signature valid for the sum of their public keys. This is nice for k-of-k multisig because now I can say you guys need to sign off to sign this, we get your public keys, we add them up, we compute an address, and now any money sent you guys need to sign because there's no other way to come up with a valid signature. It goes even further, I'm not going to go into too much detail here-- I did a presentation about a year ago about this -- where even if you don't have k-of-k situation but any other policy about what combination of policy of keys can sign, well you just need a merkle tree and you build a tree where every leaf in the tree is a combination of the keys that can sign, and the merkle root is your address, and when signing you give the root and the signatures on the leafs and so on, and this has in practical cases it's pretty much constant time for verification.

Cancelation

Unfortunately there is a big problem with this. Again this is challenging our assumptions. As I told you, you can create a signature valid for the sum of your keys. So you tell everyone your key Q1, but your actual key is Q2. You don't say your key is Q2. You say your key is Q2 - Q1. He subtracts the other guy's key from it. So the result is now just his key. This would mean that he could sign for both of them while everyone is assuming that we have created an address that is multisig that actually requires both of their signatures. This is the cancelation problem. You can choose your keys in such a way that other people's keys get canceled out. Usually this is not a problem because your keys are chosen before this scheme starts. However, in bitcoin we just have addresses, we don't have fixed keys in advance. It would be annoying to go assuming we send signatures on every address to prove that we know it, that would be one solution. So it's relatively annoying. This was for a long time a problem that we didn't know how to solve, until we discovered there is a trick that seems to make things work.

Before signing, everyone multiplies their private key with the hash of their public key. This is not a problem. The result is now that instead of adding the keys together, it's the sum of the keys multiplied by their own hash is the formula down there. It's slightly harder to sign. All the other properties remain. I started to work on the proof that this is secure. I thought I found one. But whta I found a proof for was that the same cancelation property where there is one user and the other cancels out the first one, is actually impossible under this scheme.

In particular, if oyu have an algorithm to figure out what the resulting private key would be after cancelation would be, under a 2 user algorithm, you could use the same algorithm to break Schnorr signatures themselves. So this is hard. People have studied this. Great.

Unfortunately after talking with Adam and Greg and some other people at Blockstream it turned out that we couldn't extend this proof to the more generic case. And then Greg Maxwell came up with an attack which only applies in the case where there are multiple adversaries, multiple people who can each choose hteir keys together to cancel out the first one. There is a really cute algorithm called Wagner's algorithm which would completely break this in no time.

So we need another solution. Instead of just multiply each key with the hash of itself, we multiply it with a hash based on itself and all other keys that are being used. So now an attacker cannot invent any key in the scheme anymore, because any key added to the scheme would change his commitment and break the linearity property that he used to derive. We have a sketch for a proof that this is secure. Unfortunately it would be highly inefficient for key tree signatures. Say you have a key tree with a million combinations, now for each of those million key combinations you need to do elliptic curve ryptography because eah of those would need an individual multiplier and this would kill key trees by several orders of magnitude. You don't need to commit to the exact set of keys, you could commit to the superset instead. To break this scenario, an attacker has to be able to choose their own key and not pick one from a set. We believe this is secure. As I said, I am not a cryptographer. If people are willing to help out on proving this, I would be really grateful.

Aggregation

This is something that Greg Maxwell came up with-- we can do aggregation over all signatures in a single transaction because this is the case where you're trying to protect against a situation where you don't know what's all the signers are in advance. You reveal them on the fly. You still have all public keys, we're not aggregating the publi keys in this case, we're only aggregating signatures. We would do a single validation with the verifier. This doesn't have hte largest performance advantages. It does have the batch validation speedup, though. Here is a nice graph. The blue line is the current block size that is increasing as we see, the purple line was what if we were able to strip out all signatures which is something that we could do with OWAS or BLS we could go even much further with mimblewimble. But this is shorter-term thinking. The green line is the result if all bitcoin transactions in histor would have used signature aggregation from the start. It's a 20% reduction in block size. That's not ground breaking, but what it does have is it finally gives the financial incentive for coinjoin because now the cost you bare in a coinjoin for the space occupied by signatures is shared by all the participants. The more participants oyu have, it doesn't matter; there will only be a single signature for the whole thing, and the cost for the signature is shared. It's a small advantage, but I think the subtle incentives are important I think.

Bringing this to Bitcoin

So we could do OP_SCHNORR. Instead of CHECKSIGVERIFY, with the segwit script versioning, we could just define a new version number and say that CHECKSIGVERIFY now means Schnorr signature verify. Single signature for every key. We could do better and make an alternative, called OP_CHECKMULTISIG which would have k-of-n, n keys, 1 signature. If we have the above plus a merkle check, or merklized abstract syntax tree

(MASTs), we could choose some combinations of keys and provide a single signature. If we go as far as doing signature aggregation, we could do pretty much anything with a single signature for an entire transaction.

Signature aggregation

Signatures right now contain the actual ECDSA signature with concanated to it the sighash type. We would change the checksig operation to either take a sighash type, or either a sighash type and a signature. So during an execution of a script, we add all the signatures and all the pubkeys and all the messages that get seen during an entire transaction into the stack, and then at the end we do the signature validation operation at the very end. After segwit would be a really easy thing to do this.

Future work

We need to do academic writeups about this delinearization scheme. I don't know much in the literature about this exact case. Waiting for segwit to rollout because we need the script versioning. And we need to write a BIP.

Thank you.

Acknowledgements: Gregory Maxwell, Adam Back, Andrew Poelstra.

Q&A

Q: Fixed size signatures?

A: Yes. We pick a scheme with 64 bytes period for signature serialization.

Q: Only one type of sighash?

A: It's in fact compatible with multiple types of sighash types, but it's not compatible with multiple signers being online at the same time. There's a solution for this, I didn't want to go into the details. You can say we add an extra byte to define several registers and the group of signers that are online at the same time say sign in this register, and there's maybe a second register for... you lose the benefits of the complete aggregation but it still works.

References

Schnorr signatures in bitcoin (such as OP_CHECKSCHNORRCHECKSIG), and tree sigs and poly sigs

Schnorr signatures and BLS signatures

tree signatures (slides)

libsecp256k1

https://github.com/WebOfTrustInfo/rebooting-the-web-of-trust/blob/master/topics-and-advance-readings/Schnorr-Signatures--An-Overview.md

https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/

Discreet log contracts (2017)

http://diyhpl.us/wiki/transcripts/discreet-log-contracts/

Tadge Dryja

video: https://www.youtube.com/watch?v=FU-rA5dkTHI

paper: https://adiabat.github.io/dlc.pdf

slides: http://bit.ly/2uSqkV6 or https://docs.google.com/presentation/d/1QXZBtELcVMoCq6wx-rJr31KvtsqxxcWIewMvuSTpsa4

https://twitter.com/kanzure/status/902201044297555970

We are going to have next month on September 13 when Ethan Heilman will be talking about something controversial but he hasn't told me what it is yet. But it's controversial. I am Amy. I organize the meetup with some help from other people in the MIT Bitcoin Club. I write about blockchain technologies and stuff. I have written for coindesk and bitcoin magazine and now I'm writing for Forbes. It's a great space to have fun in. I used to come to meetups and not know what anyone is saying, if you're not sure what you're hearing, keep coming to the meetups and it will eventually all come together. Tadge will be talking about discreet log contracts, which are smart contracts on bitcoin that you can't even see. We're going to meet at mead hall afterwards to socialize. Beer. Segwit.

Introduction

Hi everyone. I'm Tadge Dryja. You can see this. I'll try to be loud. If you don't hear somtehing, or have questions, stop me. If I go too fast, I sort of made it so that the stuff at the end is not so important. We can run out of time and that's okay. I will start.

I work at Media Lab which is sort of over there I think. Right? That way. At the Digital Currency Initiative which is a small lab in MIT Media Lab that works on cryptocurrency stuff, cryptography, blockchain tech, we have the director over there. I guess the primary thing that I work on is lightning network, which is something I coauthored baout two years ago. This talk is a branch out of that. It's similar. I'll explain. I also work on other fun bitcoin stuff.

About today's topic, it's a new thing. I wrote the paper about two months gao. This whole thing might not work. I think it will. It's new stuff, and maybe someone will read it and give me a mathematical reason why it doesn't work at all. But so far, I feel like the people that know this stuff better than I do have given their OK so far. It's basically smart contracts (whatever that means) using bitcoin, similar in structure to the lightning network. Nobody can see the contracts.

I'm not as familiar with.... I know how ethereum works, but I haven't programmed a lot of ethereum stuff, even though I did write a part of their proof-of-work algorithm. That's the reason why all the graphics cards are expensive today. I know. So it's also a pun because of discreet (quiet and unintrusive) where you can't tell if it's a smart contract, and discrete where it's non-continuous where you have integers instead of reals. And discrete log math is what a lot of bitcoin math is based on. If you don't know about the elliptic curve discrete logarithm problem by now, then by the end of this hour you will definitely know about it.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=4m15s

Outline

I'll talk about the structure of the lighting network, for people.... how many people here have heard about lightning network or sort of know how it works? Okay, most people. And probably fewer know about homomorphic additive that goes into it? I'll talk about elliptic curves, combining keys, smart contracts and the problem with oracles, schnorr signatures, and this semi-new thing that I came up with called anticipated signatures (using schnorr signatures), discreet log contracts, scalability and privacy and then uses. I'll try to stay within an hour or 45 minutes.

Lightning network

The idea of the lightning network was about scalability. If you want to make a bunch of transactions in bitcoin, you have to broadcast a bunch of transactions to everyone which clogs up the network and costs fees. To interactively make a transaction with a single counterparty, the idea is that you spend to a 2-of-2 multisig where both parties need to sign to spend the money. That's the person you're locked in with. You keep sending messages back and forth to re-allocate money in that channel. It's enforced in the bitcoin blockchain in that only the most recent transaction is valid. This is tricky because the global network doesn't see the transactions, only the channel participants know about those transactions.

So the idea is that you, someone, let's say Bob in this case, Bob creates a fund transaction and here's an output for the transaction and there's 10 coins there. From the network's perspective, it sees that someone sent 10 coins there. And then Alice and Bob together can create spends from that. So they both need to sign to spend the 10 coins. They exchange signatures saying well okay Alice gets 1 and Bob gets 9. They both exchange signatures. Whenever they feel like it, they can broadcast this state 1 transaction. But they choose not to because they want to keep the channel open. So then they say let's switch. Bob pays Alice. The difference between state 1 and state 2 is that Alice gets extra coins. So Bob sends signatures and they create that. They can also go in reverse where Alice gives Bob some coins. So from the perspective of the network, all that the network sees is "here's 10 coins". But for Alice and Bob, they agree to the state of the channel under penalty of a lock-up period and commitment of the last agreed (mutually signed) state. They sort of only need to keep track of the most recent state. Old states are valid to the blockchain but Alice and Bob know that the old states are invalid because they are old (and they have more recent transactions that invalidate the older transactions). So to make the network aware of this, they must broadcast the most recent transactions of course.

This is enforced with an output script where basically you're sending to 2 keys. Although I say this is Alice's and this is Bob's, that's like who owns it. But that's not who the network specifies, exactly. The output script is quite simple. It's just key X or key Y and time. So there's two keys that can spend this money. One of the keys can spend it immediately. The other key has to wait, let's say waiting one day before they can spend the money. In lightning network, when you close it, let's say Alice closes this transaction, it's actually Bob's key plus some other stuff can spend it immediately. But Alice's key has to wait a day before she can take the money. She can close the channel but she has to wait a while. The wait is to give Bob a chance to grab it in the event that Alice broadcasted the wrong thing. If she broadcasted an older state, she has to wait some time to get that transaction committed, but Bob can broadcast another transaction immediately because Alice's transaction was an old state. The immediate key is the combination of both Alice's and Bob's key. So when they switch to the next state, Alice sort of rescinds her claim on the previous state by giving Bob the ability to immediately grab the latest version or state of the channel.

Elliptic curve cryptography

And how do they immediately grab it? How do you have this combination pubkey X? I'll go into how this combination process works. So elliptic curves. This is the basis of all the signatures and fun stuff in bitcoin. Not the proof-of-work, which is just a hash function. So you have these points on a curve that look sort of like this. The bitcoin curve doesn't chop into two pieces actually, because it's actually $x^3 + 7$ instead of $-4x + 2$. So none of them actually look like this because it's modulo some giant number, so it all looks like random dots. But it's much easier to explain and look at, like this.

The idea is that you've got these points on the curve. You have defined addition. You can add p plus .... well, you can double a point by taking its tangent. And you can add two points by drawing a line between them. So you draw a line between two points, find where it intersects, it will intersect at some point, and that's the sum, the inverse of the sum. You have these points, you can add them, you can't multiply because that's not defined. But you can multiply them by regular numbers. You can say this is 2p and this is 3p, I can add 2p to p, I can have a regular number coefficient for these points but I can't multiply or divide the points. The rules of this system-- and htroughout this I will use lower case letters to represent regular numbers, scalars, numbers, and uppercase letters to represent points on the curve. With regular numbers you can add, subtract, divide, multiply. With points, you ca add and subtract but you cannot mulyiply points. Addition is defined, but not multiplication. And when you mix the two, you can multiply a point by a regular integer, but you cannot add a point to a regular integer, that's not defined.

And also, in this curve, you have some curve that everyone agrees on that we use in bitcoin, and you have a generator point G, and this is the point that everyone starts with. And public keys are some secret number multiplied by that generator point. So it's kind of fun because while you can multiply, you can now... point A is some coefficient times G, but we don't know what the coefficient is. You can't divide, to get the scalar back out. There's no real way to do that. And it's homomorphic, so what you can do is you can say this coefficient times the generator, so this is a point-- a coefficient times a point is a point, and do it again to another point and coefficient; it's the same as adding the integers themselves and then multiplying a point. So this homomorphic property is very useful and it's the basis of what we're going to be doing here.

The sum of the private keys... if you're from the software engineering point of view, if you add the two private keys-- a private key is just a 32 byte sequence; if you add those two private keys, it will be the same as if you add the two points in the pubkeys. So that's fine.

Revocable public keys for lightning network payment channels

So what you can do to make the revocable public keys in lightning network, say Alice has public key A, and Bob has public key B, and they say okay the combined key is just the sum of these two points. We add A and B, we draw a line between them. Alice knows secret key A, Bob knows secret key B, and neither party can sign with secret key C. Neither party knows the sum of these two. But if Bob reveals little b to alice, then Alice knows how to sign with C. She can just add a and b together and then they know how to sign with C. The inverse could also happen, but in this case, one party reveals a key to another party and the key is now known.

Alice gives her part of the key to Bob to revoke her claim on the transaction. So this public key x is a combined key. Public key Y is Alice's only. So this is Bob and Alice. Alice reveals to Bob. So now Alice has rescinded her claim on this. The fact that she has to wait means that if Bob sees this on the blockchain then Bob just grabs it immediately. So it's going to be gone by the time that the timeperiod has been exhausted. So it's this kind of weird thing where you do need to monitor the network. If you're asleep at the wheel, Alice might be able to broadcast something old that shouldn't be broadcasted, and if Bob doesn't see it in time, then Alice might be able to get away with it and spend it because Bob didn't broadcast the alternative quickly enough.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=14m

Smart contracts

This is a smart contract. It's fairly straightforward. It's a payment conditional on something. In lightning network, the smart contracts are only based on internal data. It's Alice and Bob exchanging data which determines who gets what money. But in smart contracts, we're interested in external state.

If we need an external state, then we might want an oracle. In this example, Alice and Bob want to bet on tomorrow's weather. So if it rains, Alice gets a bitcoin. If it's sunny, then Bob gets a bitcoin. The problem is that the bitcoin blockchain doesn't know what the weather is. There's no OP_WEATHER. Do people have questions on the structure of lightning and how you enforce only the most recent transaction is valid?

More fancy is what if we want a smart contract based on something external to the blockchain? Okay, we need an oracle. You kind of need an oracle. If you don't have an oracle, then you could say that well Alice and Bob could just agree that if it rains, Alice gets a coin, and if it's sunny then Bob gets a coin. If they are totally chill with each other, then it works great. But the point is that bitcoin is the enemy of currencies, you don't trust anyone and everyone is trying to kill you or something. So just having 2-of-2 multisig, it doesn't really work for this problem. What you can do is let's say it rains, and Alice says okay Bob you have to sign. And Bob can say that well I don't agree with that-- it didn't really rain, it was barely a drizzle, so I'll give you 0.7 and I'll get 0.3. But Alice will contest that. You need some sort of third-party to step in.

A third-party could decide in the case of conflict. So you get a 2-of-3 multisig oracle. Instead of sending the deposit to a 2-of-2 multisig with the two contract participants, you send to a 2-of-3 multisig. You have 3 keys, Alice, Bob and Olivia and Olivia is the oracle. If they are both chill, then they both sign without contacting Olivia at all. If they are both chill then they do 2-of-2 and they ever contact Olivia. If they are fighting over it, and they have used 2-of-3, then they ca turn to Olivia and she will decide in favor of whatever correct party. You only need two signatures, so the conflicting party is left out. Well, the problem with this is that say that Alice calls up Olivia and gives her a bribe. Alice has lost everything and she might be able to bribe the oracle. The oracle had to be part of the setup process. So Alice might collude with the oracle.

It's interactive. The oracle sees everything. They also decide the outcome of every contract. They can equivocate. They could say it's sunny to one contract and it's raining to another party. So the oracle knows, but it's really hard to prove. You could have some kind of proof that this signature represents rain and it didn't rain, but it's on an individual basis. The interactivity is ugly because you can bribe them and try to influence the oracles.

Having a system where the oracles can't equivocate and can't see what's going on... it's not a cryptographic assurance; maybe they can still collude anyway, but it's a lot better of a system. How do we do this? How can we have the oracle sign off on outcome sand effect the state of these contracts, without them even realizing that they are doing so? There's a nice way to do this.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=18m40s

Schnorr signatures

I am going to talk about Schnorr signatures, which is also relevant to bitcoin. I think people have talked about Schnorr signatures and how they are cool and how we want to put this into bitcoin in the future. Sort of random aside, ... schnorr signatures, one random thing; Schnorr is kind of a jerk. Hopefully this guy doesn't sue me. This guy patented this signature scheme. It's really a straightforward signature scheme. In writing my paper about discreet log contracts, I accidentally derived the concept of Schnorr signatures but I didn't realize it. I was told by other people that this is a Schnorr signature scheme. So it's an obvious thing to do, and he patented it, and nobody used it for 20 years, and this is why we have ECDSA instead. If you know the math behind ECDSA signature scheme, it's ssort of ugly, and it was developed that way so that it wouldn't infringe on the Schnorr signature patent. How much, how little could we change so that we're not infringing on the patent? It got pretty ugly. So we're probably not going to call them Schnorr signatures in bitcoin, we will probably be calling them Bn signatures. There's another team that wrote something similar. So, yeah.

In signatures, you don't just have your public key. You have other things going on, like a message. In this explanation, you have a scalar and a point, that's your public key. You come up with a random key, there's your public key, H is a hash function probably sha256 in this case, and hten there's some message which is just some bytes. Schnorr signature is real simple. You have a public key. You have a random scalar k, you multiply it by the generator point to get a point r, and this is similar to a public key but you're using it for only one signature. And to sign, you just, for the signer it's really simple, you just say your signature s is k minus the H(message) and r times my private key. That's it. It's a little weird in that you concatenate your message, and r is a point, and you can have an x coordinate and a y coordinate and hash that together. Little k is a secret. The signature is r and s. The signer reveals r and s, but the signer keeps k and little a secret. It's important to keep both of those secret. To verify, you have r and s, everyone knows G, you know big A because that's the public key, and you know the message and the signature. To verify, you need to make sure that s is really equal to K minus H(m) * r * a. You don't know little k... but you know k times G and little a times G. Because that's just r, and big A times... because that's their pubkey. You algebraically multiply both sides by G, so you have s times G equals this whole hash times a * G, and ... H(m) ... you know these values, so it's verifiable. That's a Schnorr signature. You can't forge signatures because you can't inverse the hash function and r is in both sides. You can't just compute a scalar that looks like that. r is in both places. If you can break the hash function or figure out the discrete log then you can forge the signature.

In my discreet log contracts scheme, I am introducing a fixed r value signature scheme. Generally what you do is you say I've got a pubkey A, and my signature is r and s, and you verify it with this formula. With discreet log contracts, we say that the pubkey includes an r point. And the signature is only s. It's the same formula. Everything is the same, except that I pre-commit to this r value. Instead of coming up with a random k at signing time, I come up with random k at the time of revealing my public key. It's the same thing except now you've got a couple of differences. The big difference is that you can only sign once. When you're signing, with the same public key, you need to use a different k each time. If you don't, and this is a little ugly, I'll go over it real quick: say you have two signatures on two different messages with the same k and the same pubkey A. You can subtract those two signatures, so s1 = k - H(m1)*A, s2 = k - H(m2)*A, so the difference between these two, the k's disappear, so you're left with these two hashes, you can factor out little a, and you can solve for a. So you know s1 and s2, you know H1 and H2 are, and you just revealed the private key. So don't do that, always use different k's. There's a standard, rfc6979, where you can make k deterministic on the message you're signing, but you can't do that in discreet log contracts.

This is a fun fact. This is what brought down the playstation3 code signing. They used the same k for all their code signing. And so people were able to figure out their private keys and sign code and private video games. That's probably the first time that I learned about elliptic curve private signatures. That was like 2001 before I really got into this stuff. But I was like hey that's how they hacked the playstation.

Anticipated signatures (fixed r value Schnorr signature scheme)

So with this, if you already know their point r, and their public key, that's not normally the case, but in this case we're going to pre-commit to an r value. If you know a, r, and a message m, you can't compute s, but you can compute s times a generator point. That's just going to be r minus this hash times their public key. You know all these things: G, r, the message, their public key. You can compute, you can pick a message and compute what their signature will look like times the generator. You don't know the scalar, but you know the point it will multiply to. So you can compute sG for any message without knowing s. This is not how I cam eto this, I came to it backwards. It makes more sense if you're doing it this way, though.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=27m

You have this signature which is sort of a key... you have this unknown scalar, but you know what it is times the generator point. It seems like a key pair. It's the same as a public key where like okay I know what it is times the generator point, but I don't know the actual scalar to get there. So you can use the oracle's signature as a private key in this case. Similar to lightning network, you have a bunch of states spending from a single transaction output. But instead of making them sequentially... like in lightning, where state 2 is made after state 3; in discreet log contracts, you can number them but it's arbitrary because you make them all at the same time when you're creating the contract. Instead of saying the most recent transaction is valid, the validity is determined after the fact by the oracle. So the oracle will contribute part of a private key to one of these outputs. So say you're making a transaction about weather betting, where some output goes to a different party. How do we enforce that only this one should get broadcasted and this other one shouldn't get broadcasted? What we do is that the oracle's signature is thrown into these private keys. Olivia's signature s, treated as a private key, we don't know what it is yet. Olivia hasn't signed anything yet. We're building the contract. She might sign the eclipse message, the sunny message, or the rainy message. We can calculate s * G for any given message we expect her to might sin. So we take the contract public key for Alice which will be Alice's public key plus s * G, and her private key will be Alice's private key plus s, which is the signature that Olivia will provide. We don't know what the private key is going to be, unless Olivia signs. So say there's three messages that could be signed, like sun, rain, eclipse. So there's three signature keys. The detail is going to be H(word sun). So we can make rain public keys for Alice and Bob and sun public keys for Alice and Bob, and so on.

Q: ... how can... class of outcomes..

A: Yes, in this case, in all the cases actually, we're assuming there's only one possible outcome. It could be sunny and have an eclipse. It could be rainy and there's an eclipse. Alice says that she is going to sign one message-- she can't sign more than one, so if she signs two, she reveals the private key. Only one of these is going to be valid.

Q: Okay so go back to... I'm sorry, can you go through the transaction history of setting up the contract?

A: Yes, sure. The sequence would be that you build these first. First, Olivia commits to an r point and says this is the keypair I'm going to use to sign tomorrow's weather. She should probably be specific about what possible messages she could sign. You want everyone to know that there's only three messages or whatever. It could be signing the temperature in celsius and there's only 100 possible things I could sign. Alice and Bob need ot anticipate every possibility. So for each possibility, they create a transaction and hand each other the signatures. They say I'll send 9 here and 1 here in this case, and this key is Bob's key plus Olivia's sunny signature, and this one is Bob's key plus Olivia's eclipse signature, etc.

Q: How did...

A: That hasn't happened. Olivia has not yet signed. Olivia just said here's a pubkey. I'm going to sign a message tomorrow. You know a, you know r, and you can anticipate the messages she's going to sign. So you can calculate the signature times the generator point, but you don't yet know the signature. So Olivia commits to these things. They can build the transactions. They can fund the transaction output and send money to the contract. And then they wait. They have to wait because if they broadcast these, they wont be able to spend from it. They need Olivia's signature to spend it. It says Alice but they can't grab it on its own. So they need Olivia's rain signature to broadcast this or to get this.

Q: Am I right in understand that the main reason to do this instead of many hashlocks is to enforce that Olivia only sends one signature?

A: Okay. So yeah. You could have a table of all possible hashes and preimages. There's a couple improvements in this case. One is that Olivia can't equivocate... she can only release one, without revealing her private key.

Q: Is the assumption that Olivia is using a long-term keypair that she wants to maintain?

A: If Olivia is a known entity, A can stay the same each time. Or without that property, you can have a parent key above that which endorses subkeys, and so on. One is that it's harder for Olivia to equivocate. It saves space. What happens if there's a million different outcomes? Well Olivia only needs to commit to one 32-byte r. It also helps in that if they end up on the blockchain, you can't tell. If everyone sees that Olivia is committing to different hashes and they're going to reveal a preimage, that would show up on the blockchain. They will say I don't know who these two people are, but they were betting. Whereas in this case, they can't see it.

There's three possibilities-- three different signature keys-- you can add those. It's the same script as in lightning. You have these values.. but it's swapped. In lightning, the correct use is the timeout. If you broadcast the right thing, you have to wait. But if you broadcast the fraudulent thing, your counterparty will immediately grab it. However, in discreet log contracts, the fraudulent one is the timeout one. Whereas if you broadcast something and you don't know the immediate combined key, then it means you broadcasted the wrong thing and the other person can grab it later.

So they build this. They build these. They fund it. They wait for Olivia to sign. So it rains. Olivia signs the message 'rain'. Alice or Bob, but Alice is winning so she probably does this. So he knows the value, she has the private key to sign for this one now. Alice can broadcast state 2. Once she does, she knows the private key of course, she can sweep it and send to her own address, which she wants to do immediately because otherwise Bob will eventually be able to spend from that output. So she broadcasts state 2, and then also broadcasts a spend from that. It's a little ugly because you have two transactions that are redundant and they both have to go on the blockchain, it's not so bad. The software can do it at the same time. If she doesn't broadcast the sweep, then after the timeout then Bob would be able to grab them.

The timing is a little different from lighting. It's sort of better. In lightning, you don't know when things are going to happen. You have to be vigilant and watch for fraud constantly. In discrete log contracts, you broadcast the output state 2 and you immediately grab the funds from it. The only times that you will have to wait is if something wrong happens. If Bob broadcasts state 1, that's annoying for Alice because she can't get her money right away, but she can get it eventually, but Bob will never be able to spend it because Olivia is never going to sign that outcome. The only reason to do this is to be a jerk. So if Bob knows that he is losing and he is going to lose all his coins anyway, he might broadcast the wrong state, and this will inconvenience Alice. If Bob tries to be a jerk and delays Alice, then he should lose everything including that extra 1, and Bob should lose that too, as a disincentive to be a jerk.

Q: What stops Olivia from colluding with Bob or Alice?

A: It's still possible, right? The biggest aspect is that Olivia has no idea any of this is happening. Olivia says on some website or somewhere, here's an r point, I'm going to sign the weather tomorrow. She has no idea that Alice or Bob are doing this. So it's a little harder-- they are not already in contact with each other. If they are a good oracle, then they might say hey I don't even have an email address. Olivia could collude, but it will be public and they can only collude one way.

Evil Olivia-- a bad oracle can cause contracts to execute them the wrong way. But they can only do this once. They can't sign two different outcomes. Anyone can compute the private key and sign any message at that point, and it would be totlaly up for grabs at that point. An incorrect signature is public. If olivia is well known and they keep reporting the weather every day and it's been okay, and any invalid or wrong signature, everyone will be able to see that. It's possible, but it's a bunch of things that make this hopefully unlikely.
Q: What do you mean people will know if it was invalid? If I don't know if that contract is..

A: You don't have hte contract, but you know that Olivia said that she was going to sign something, and you have their sunny signature, you... this is like. Olivia doesn't even know the blockchain exists. She just says point r, here's my signature tomorrow, anyone can be given this data. Regardless of the contract, everyone can see that they made a signature for the wrong thing. Once they do this, everyone can post that signature and your software could immediately detect that the signature was for the wrong state.

Q: In that example, you can tell what the weather is. But say the contracts are.. less public data.

A: I'll talk about more realistic contracts, sure. It should be something that is non-ambiguous. Best case, it's a number that everyone is going to agree on. The oracle is there to just grab the number, sign it, and then you're done. But if it's something where people are going to argue about it, then people aren't going to be able to do this. This works better for pricing data, like the price of oil. Where there's no real argument. The oracle can just broadcast it and maybe thousands of people are using it and the oracle doesn't know.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=42m

Scalability

The whole process is three transactions. You fund it, then you broadcast a close, then you sweep it. However, if the parties are chill, they can reduce this to two transactions, where there's a funding transaction, then you sweep to the correct party. If they both agree, and they are cool, they can connect to each other and say hey I have the rain signature and you know I can broadcast it, how about we just sign a regular 2-of-2 multisig where I get it in the clear and you get yours in the clear, and then enough to sweep. So this is a "gg" transaction if you lose (good game). That should save some space. That helps a little.

A more advanced thing, which is more complex, is that you can make a discreet log contract within a lightning network channel. So if this works, zero transactions go to the blockchain. It's sort of like adding an HTLC, you're adding a third sort of output that happens to be 2-of-2 multisig and from this output you build the contract. It does't even exist on the blockchain, it's just in your channel state, and if you want you would have to close both, if the parties become uncooperative you broadcast the three outputs and then from that you are broadcasting those 2 outputs, and then sweeping. So it's a little ugly when it doesn't work well when people are uncooperative and you have to go on chain. But if they are cooperative, then there's no additional transactions that need to go to the blockchain. So for parties that are cooperative, this is highly scalable.

It's sort of a threat: Bob knows that Alice can broadcast this, that she can close the channel without asking, and take coins. So if Bob is a good sport and knows that this is possible, then they will agree to update the balances and they can do many different bets all at the same time. It's a little harder because you need to put timeouts and stuff so that people don't broadcast old states, but it should work.

For the in-channel contracts, nobody sees that they occurred. Even if you broadcast to the network, it's not clear that it was a contract. It looks like a lightning network channel got opened and closed. So it's not clear that you guys participated in a smart contract. Even the oracle knows s and stuff, you can't tell because it's a new random pubkey that has been added to one. It's kind of cool. In practice I think you could get some probabilistic idea of which are smart contracts and which are lightning channels. You could see lightning channels ending with timeout, but these will end with the key script, but t you won't be able to know actual details.

https://www.youtube.com/watch?v=FU-rA5dkTHI&t=46m

Use cases

So, real use cases. Weather is great and all, but you probably want something more fun or interesting. And so what people like to do is talk about money and stuff. Probably a big case would be betting on the price of the dollar. This is a little backwards seeming at first glance. In this case, you have to think of bitcoin as money and the dollar is just some altcoin. You price things in satoshis. So a dollar is about 25,000 satoshi BTC. You could make thousands of transactions based on the price of the dollar. So the oracle will sign in the price of the dollar tomorrow, in satoshi BTC, so if the price of a dollar is 1 satoshi, then one counterparty gets one, the other counterparty gets the other side of the deal. Zero means that the dollar is worthless and bitcoin has taken over the world.

You can make, you know, 10,000 transactions or 100,000 transactions which will be about 10 megs. It's off-chain and 2 computers can make 10 megs of transactions, that's no big deal. This is a little ugly because it's 10 megs and also, in practice, probably the range doesn't need to cover the whole thing. There might be ranges where Alice and Bob say hey if the price of bitcoin goes up to $100k, or $200k or whatever, you win. Whether it's higher, then Bob gets all of it. Conversely, if the price of bitcoin tanks, then Bob should win. You can split up the r values into two different r values and one is an exponent and one is an antica. So you can do the order of magnitude, and then the number of significant digits. It saves a lot of space for the off-chain part. It's not just scalable on-chain, but also scalable off-chain where you can get down from 10 megs to 1 meg. Which is overkill because 10 megs is not a big deal.

Multi-oracle

You can have multiple oracles, let's say that they want to use oracle 1 and oracle 2, that's easy, you just add up their points. You can add these into the pubkeys. It has to be both signing the same thing. And there's no size increase. You can do m-of-n. We need two of these three oracles to sign it, but it starts to blow up the size of the state between the two parties, it gets really big really fast. The other tricky part here is that they have to sign the same exact thing. If one oracle signs 52 and the other one signs 53, and they didn't intend to sign different values, then you can't close your transaction because none of your things will add up. You probably also want a timeout transaction where if the oracle goes down, then after a week or two, Bob and Alice should each be able to get half of their money back. If the oracle dies then it's a wash trade and they should revert.

Other uses

Currency futures are probably the ideal use case. It's a synthetic future. It's sort of a contract for difference. I don't really care as much about the volatility, say, and the opposite side is that you have super bitcoin that are much more volatile then if the price goes up then you get more, but if the price goes down but you also lose them. And you will find people that want this, because people love leverage and gambling. It's nice because you can get a stable dollar-based currency and you just need to find the people that want to make that deal.

You can do stocks, synthetic assets on stocks. You can do a share of apple, if the price of apple goes up or down in bitcoin then the contract reflects that. Insuance is just gambling, there's no difference. Olivia could report on whether a house burns down, or whether a hurricane hits, then people can create insurance contracts. It's just general conditional payments. It could be on numbers, but really any pre-determined set of possibilities where one element of that set is the thing that happens.

It can't be things like "what's the name of the highest grossing film of 2017". You don't know the names, so you can't build that contract. It's borderline because you sort of know the movie sin production, but who knows, maybe something comes out left field and it grosses the highest, and you didn't know the name, so maybe that's a mess-- but you could do a timeout in that case really.

You don't need a token-- you can just use bitcoin. It's permissionless. Nobody can even tell what you're doing. You can do it on any blockchain. You can probably do it on ethereum, but I'm not doing that. There's no token you need. I am not doing an ICO. Sorry. I won't make 100s of millions off of this.

I think this is better than Augur. I live three blocks from them. They have raised millions of dollars to make a prediction market but I don't think they have finished it yet. There's gnosis, delphi, there's a bunch that want to do smart contracts or whatever. But I think my scheme is better because it's harder to equivocate, more private, things like that.

Questions

Q: In terms of currency futures and commodities... would this be a replacement for an exchange? Would an exchange adopt it?

A: The idea of an exchange.. there's different elements of an exchange. They are a place for buyers and sellers to find each other. Also they have custody and they execute on their behalf. They also have custody. How does Alice and Bob find each other? There's still a need for some kind of meeting place. An exchange could do that, and they could offer a trade and then people who want to buy and sell could find each other on that exchange and create those contracts. But they don't need to keep custody of people's funds in this model. Or the exchange could be a counterparty where they say hey we have a ton of money and we will take one side of your contract bets, and we even it out, we have a lot of capital, there's a spread and we enter into different contracts with different participants. But it doesn't solve how you find people for trading. Decentralized exchanges are really difficult and there's lots of timing problem. So this could mesh well with an existing orderbook and matching engine.

Q: Seems like for a prediction market, you need oracles. Augur and others.. workaround for multi-oracles?

A: They had this thing where everyone is an oracle. I know how Augur worked, where everyone voted on something and the truth is determined by majority vote. I think this is dangerous because they have this token with reputation points and the majority of the reputation endorses an outcome. But if you managed to corner the market, and the minority loses their stake to the majority, so if you managed to grab it, then you could kind of say, I'm going to sign on-- what's vote on pie. Tomorrow I will sign pi, and then you say 3.14 well nope it's 7 and the majority endorses 7... and all the honest people lose the reputation to the dishonest people. So you can't recover from this some sort of dishonesty problem. In discreet log contracts, it's up to Alice and Bob to pick their oracles. In practice, I feel like the oracle job is a natural monopoly where you're not going to have hundreds because the costs in terms of thinking about it, because hey I want to enter into a contract with you, and you say hey what about this other oracle... well it bifurcates the market, which is already tricky because of the epxiry date. It's like a options market where you have the option to buy and sell. Having multiple oracles is going to end up with a couple big oracles like Bloomberg or Reuters or something like that.

Q: It's just like the regular market. There's ... go to..

A: But if you're making a contract where you have a futures contract based on the price of oil in 1 week, everyone is going to use a specific intermediate. You have bushing, WTI, you have a few different ones, but not millions. There's probably going to be single digit or double digit well used oracles for different things. So people will mostly use the defaults. The oracle has no idea if anyone is using them... maybe they broadcast isgnatures and maybe someone uses them or not. Not sure how they get paid. I think an exchange would be a good oracle. If kraken wants to sign the price volume weighted average for their price for today, you can trust them because, they got everyone's money anyway, if they wanted to rip everyone off they could just run away with the custody anyway, so you might as well trust them to report an accurate price. They use this to drive people to their exchange as a service. We sign the price of the coin on our exchange and people can use this for derivative contracts. So maybe exchanges would use this. I'm writing software onw. I'll do it. It's pretty easy to just sign stuff.

Q: Do you envision ... so two counterparties doing a trade, based on something happening like price of oil, do you....

A: The oracle can't tell if anyone is using their data. Probably where the value gets captured is matching the users. So when Alice and Bob want to find each other. Bloomberg where you have an orderbook, you have a matching engine, they find each other, that can capture a fee maybe. It's hard for the oracle to capture the fee. They don't know who's using it. Maybe they can charge for the signatures.. but one person can subscribe to that and put it on pirate bay. You don't even need pirate bay, the signatures are 32 bytes each. I think it's not a money-making thing. I think people just provide it as a service:

Q: Is it... added.. or ..

A: The script? Okay, so, this stuff you need segwit.

Q: You need segwit there. Do you need a new opcode?

A: Nope. It's the same as lightning. So, segwit is locked in and it will activate in a week. But actually it needs segwit less than lightning does. You know the timing of everything. Without segwit, you could have a funding output that has a 2-of-2 multisig OR checklocktimeverify and a refund. But since segwit is going to activate, the script itself..

Q: I thought bitcoin only supports..

A: The Schnorr doesn't happen on the blockchain. The schnorr signature happens with Olivia signing. But Alice and Bob uses ECDSA with the keys they have derived. So the Schnorr signature doesn't touch the blockchain at all. It's the same scripts.

Q: You presented this as .. schnorr, but you can do it with ECDSA?

A: No. The Schnorr signature is required, but it happens outside of the blockchain. Olivia just says here's a pubkey and a point r, I'm going to sign the price of something tomorrow. If someone wants to use that on a blockchain, cool, but I don't know about that. That's where the schnorr signature occurs. Alice and Bob can incorporate that data into their ECDSA pubkeys. It goes into the pubkeys that can then be used for ECDSA or Schnorr. The Schnor happens off-chain.

Q: Schnorr generates the key.

A: Yeah. You generate the key with schnorr. You can't do this with ECDSA offline. You need that property of the schnorr signature. It's nice because it's not a consensus rule for bitcoin, you can run it today, well you need to write software. I started a branch in lit. I don't know how long it will take. There's no required consensus rules for this.

Q: ... have you read the plasma paper.

A: Yeah. It's sort of vague. It sounds cool. I haven't worked with Joseph in a while.

Q: Will you post the slides?

A: Sure I'll post the slides. The DCI slack or the meetup page? Sure.

Q: We have a slack. Not the DCI slack. Ethereum developers slack channel.

A: Eh, I'll post on the meetup page. Ethereum slack, I dunno. Nah, we're all friends. Any other questions?

Q: We're all going to mead hall. September 13 is the next meetup. Ethan Heilman will be speaking about something confidential. No, controversial. Well right now it's confidential because he wont tell me what it is, other than that it will be controversial. Same room same time.

Okay, thanks.

Mimblewimble and scriptless scripts (2018)

http://diyhpl.us/wiki/transcripts/realworldcrypto/2018/mimblewimble-and-scriptless-scripts/

Andrew Poelstra (Blockstream)

https://www.youtube.com/watch?v=ovCBT1gyk9c

https://twitter.com/kanzure/status/954531580848082944

We're about ready for the second session of RWC. We are going to move on to cryptocurrencies. So we have 3 talks lined up in this session. The first one is on mimblewimble and scriptless scripts, delivered by Andrew Poelstra from Blockstream.

Introduction

Thank you. Cool. Hello. My name is Andrew Poelstra. I am the research director at Blockstream and I'm here to talk about a couple of things that I've been thinking about over the past year, called mimblewimble, and also scriptless scripts. Maybe you've heard of this. I've given similar talks to this a few times before at more blockchain/bitcoin-oriented conferences. This is the first time that I'm talking to a general cryptography audience. So I'll try to tailor the talk to a more general cryptographer audience member.

What is a blockchain?

As the first speaker of the cryptocurrency sessions today, I should talk a bit about what blockchains are. A blockchain here is just a list of blocks. It's a merkleized list of blocks. Each block commits to the previous one. Each block also commits to a pile of other data, called transactions. And there are certain rules about what those transactions are.

The real innovative part of bitcoin is that there's a global consensus system on what the blockchain is. For my purposes, and the next couple of talks, we're just going to take that for granted by saying that somehow everyone has the same view of the system, and we're not going to worry too much about how they are going to achieve that.

In bitcoin, every transaction has a pile of inputs (references to previous outputs), a pile of outputs, and the essentially the only rule beyond that the transactions are valid is that each input can only be used in one transaction. You can't double spend.

If you are somebody joinin the bitcoin system, the design goal is for the system to be trustless (publicly verifiable). Anybody can download the blockchain, the series of blocks and all transactions. They validate all transactions check that each one is legitimate and has all of the necessary signatures with no conflicts. After doing htis process, they are left with a set of unspent transaction outputs, called the UTXO set. And basically every single cryptocurrency that you here about has some form of this model. In bitcoin, the system state is described by the unspent transaction outputs, those are the coins that can be spent in future transactions. In zcash or monero they use a different model where they track spent outputs because the unspent ones are not publicly visible. In ethereum it's more complex, but the idea is the same. You download all the transactions, you verify them in a row, and you're left with a state that you know came from a legitimate history.

Mimblewimble

http://diyhpl.us/wiki/transcripts/sf-bitcoin-meetup/2016-11-21-mimblewimble/

Mimblewimble was proposed a year and a half ago by this character Tom Elvis Judesor. It's a slighlty design. People can verify the state of the system without downloading all of the background data. Somehow it's possible to compress out all transactions. A lot of the data is redundant, and you don't need it to get full public verifiability.

One quick note. I said "this character Tom Elvis Judesor". We're in Switzerland. Maybe there are some French speakers that recognize this, but it's the name of Voldemort in the Harry Potter books in the French translation. It's not a real name, as far as I'm aware.

This mimblewimble system was originally proposed in the form of a text document that was published on a tor hidden service and dread-dropped on an IRC channel about a year and a half ago. That's where all of this comes from. The author just dropped it, signed off a minute later and has never come back as far as any of us are aware.

Confidential transactions and pedersen commitments

https://www.youtube.com/watch?v=ovCBT1gyk9c&t=4m45s

Let me before I begin to explain mimblewimble, let me explain confidential transactions. It's a scheme devised by Gregory Maxwell a couple of years ago where you have a bitcoin-like system but all of the amounts are hidden. The amounts are replaced by homomorphic commitments. In particular, these pedersen commitments whose structure I've displayed here. And the nice hting about these commimtnets being homomorphic is that people who download these transactions can verify what they need to verify about these amounts using the homomorphic property. In particular, this means being able to verify that the sum of the inputs is equal to the sum of the ouputs. This tells the user that no coins were created or destroyed. Verifiers don't care about anything else, only to verify that the system is working properly.

As a second complication, and this is a sleight of hand; given a dollar value v and zed mod q, choose-- you can't take a dollar zero value to the bank and split it into a 10 dollar bill and a 0 dollar value. To make this system work, we have to restrict the amounts to relatively small values relative to the group order so that they will behave like positive integers. They won't overflow under any realistic scenario. And to prevent that from happening, we need something called a range proof, which I won't go into into too much detail except to say that it is there.

Mimblewimble transactions

Our pedersen commitments consist of the actual amount and also this other uniformly random blinding factor called "r", which is used to hide the amount. And in order to produce a transaction, you need to know the sum of the blinding factors for the input. And you can maybe see this by trying to think about how to structure this, and also knowing that the range proofs serve double duty in a practical system by also serving as a proof-of-knowledge of the blinding factor.

What this means is that you can't construct a transaction without knowing the secret. Well, in bitcoin you already have this rule that you can't construct transactions without knowing the secret. The rule is that you have to know the secret key associated to the public key that every output is labeled by. Why have these two secrets? Why not just drop the secondary one, the one in what's called the script signature (scriptSig) and just use the pedersen commitment blinding factors as your secret? This almost works but there's a problem.

Mimblewimble: Kernels

https://www.youtube.com/watch?v=ovCBT1gyk9c&t=7m15s

It almost works to just use the pedersen commitment blinding factors as your secret, but there's a little bit too much structure in here, in the sense that in a real transaction your inputs come from somebody and your outputs are presumably going to somebody else. These are two disparate distrusting parties that presumably shouldn't know each other's blinding factors. So if you were to just use the homomorphic property as I have described (sum of inputs is equal to sum of outputs), then you would have this problem which is that the output owner would know the sum of the input blinding factor and vice versa. You're not transferring anything, everyone involved in the transaction would know all of the blinding factors, and therefore they would have all the outputs.

The way that mimblewimble solves this is by adding a special zero-value output called the kernel. The kernel cannot be spent, which means that to the extent that a sum of blinding factors and a kernel blinding factor are known, that's useless to somebody trying to create a transaction, they have to know a sum of actual spendable outputs. And this prevents those kinds of attacks from happening. As a second feature, the kernel acts as a multi-signature key for all of the participants in the transaction. What it is, really, is that it's a sum of all of the blinding factors from each participant and then multiplied by some generator G here. You can think of this in two ways: as a sum of pedersen commitments (zero-valued output) and the other way is as a multi-signature key. These two things are complementary and that's sort of the magic of mimblewimble.

Mimblewimble in pictures

I'm going to switch now to pictures for the remainder of the first half of this talk to give you an idea of what the implications are. So here I have two mimblewimble transactions. They have inputs, which are references to old outputs, and I have outputs, which are new outputs. The rule for these to be valid is that rather than having each input having some sort of signature attached (scriptSig) we simply require that the sum of the inputs minus the sum of the outputs equals the kernel. These things are not attached to the kernel, they are free-floating. None of these thins sign for the other things. If I'm allowed to have a transaction with multiple kernels, I can combine the input sets and the output sets, and I can have one transaction with two kernels which is already sort of a neat feature. In bitcoin, we call this coinjoin when you combine unrelated transactions. You can do this non-interactively in mimblewimble because nothing is signing the complete transactions. But then, notice that the second transaction spends an output from the first transaction. Okay. If my rule is simply that outputs minus inputs equal kernel. I can delete input and output as long as they are the same. So now I have a transaction that is smaller than the original, but still has the same net effect as the original transaction. And critically, because the kernels are still in tact and each kernel is a multi-signature key from the two transactors, this entire aggregate transaction is still authenticated by everyone involved. So somehow you're moving money from A to B to C and the intermediate step is deleted from the perspective of any public verifier but you still retain the no-inflation property (that the total input amount equals the total output amount) and the authorization authentication property which is that anyone involved in any of the series of transactions that lead to this output set signed off on everything that happened, that no fraud is happening.

Here's where the cool stuff comes in. Suppose that we've got an entire blockchain of these mimblewimble transactions. In this picture, every single block is a transaction. The reason is that every transaction in a block can be combined in the way that I just described, so every block just becomes a single transaction with a set of inputs, a set of outputs, a set of kernels. There could be an arbitrary number of kernels. And that's what your blocks are. It's hard to describe these as transactions. They are inputs and outputs, and what you have is a "diff" of the unspent output set.

Now, suppose that I'm a new verifier trying to download the state of the chain. Normally in the bitcoin paradigm I download all the blocks and all the transactions in those blocks and I verify them all and so forth. But in mimblewimble, I can do that, but what I can do is download the transaction from every single block and combine in the same way as before because my validity requirement is just that sum property. And then observe that every single input except for the coinbase inputs which create new coins-- every single input is the output of a previous transaction. This telescoping sum property that I have described can be applied to the entire blockchain and everything can go away. The only thin left is the coinbase inputs (which in practice are usually implicit, like a +50 coins every block kind of thing) and whichever outputs have not been spent which is your final system state (the unspent outputs) and your kernels (every block has a pile of kernels). And this gives you the compression that I alluded to at the beginnin of this talk. It's just this property that my inputs and my outputs cancel out, algebraically.

This is pretty cool, but it makes smart contracts and other things more difficult because when I deleted these outputs, what I'm saying is that any new verifier does not see those outputs have ever existed. In principle they had no idea that they ever existed. And in particular, if htose outputs had weird requirements on them like n-of-m signers need to be signing or a hash preimage needs to be revealed or whatever, there's no way for the public verifiers to verify that those conditions have been met because the data isn't in the blockchain or see that those conditions had ever existed. So it seems like this is really a limited thing. When the paper was published in August 2016, that seemed to be where we would be left with, but in the second part of my talk I will explain that we can actually nonetheless do some cool things even though our verifiers don't see all the data.

Mimblewimble scaling numbers

Let me give some numbers about what we get from this compression technique in mimblewimble. In bitcoin, we have roughly 150 million transactions, 400 million outputs, 65 million UTXOs. If you download the bitcoin blockchain today and you maintain a full transaction index, that's about 180 GB of space on your disk. If you were to use confidential transactions with pedersen commitments rather than explicit values this would more or less triple because the pedersen commitments need these rangeproofs attached to them. With mimblewimble, the intermediate outputs go away and their rangeproofs too. So we would have only 18 GB of transaction kernels, one per transaction, 2 GB of final output unspents, just curvepoints that are 32 bytes each, and 45 GB of those rangeproofs. Despite adding confidential transactions ,you have a total system that is much smaller than bitcoin. Thanks to Benedikt Bunz and Dan Boneh at Stanford and someone at UCLA, we have a new efficient rangeproof that came out in the lats month called bulletproofs (see also). And the result of that is that the mimblewimble system with bulletproofs is much smaller than bitcoin without any rangeproofs at all, which is a really exciting development.

Scriptless scripts

Okay, so let's move on to the next part which is that, this is nice but this if this is only about unconditional coin movement then this is not particularly exciting. However, scriptless scripts, which is something that has been developed throughout 2017 and continues, a very broad research project, is a way to use these kernels and kernel signatures to attach conditions to them without modifying the system so that the verifiers need to understand new rules. What I mean by this is that a set of parties can decide on some sort of contract or protocol that they want to execute, and as a result of faithful execution they will produce a valid signature and the blockchain and its verifiers can validate that the signature is valid. The blockchain does not need to know any of the details of the original transaction.

Historically this came from mimblewimble. We wanted to be able to do cool stuff with mimblewimble, and we couldn't. But in fact, any system that supports or Schnorr signatures or some signature scheme that is linear in the data can do this scriptless script technique.

There are many reasons to do this. I described the motivation for mimblewimble, but even for bitcoin and friends this is exciting. The reason being that, the existing paradigm for doing this is to use a script-signature system where you lay out the rules and then you lay out an explicit witness that those rules were followed. And then everyone downloads all of this and then verifies the witnesses for every single transaction, which prevents the mimblewimble-style compression that we talked about, but it also means that anybody who wants to do cool stuff can only do it within the framework that the entire system is aware of, namely the consensus system where everyone agrees on whether rules were followed or implemented at all. It's very difficult to extend this to do things that require primitives that do not or do not yet exist in the consensus system. As a secondary feature, we care about privacy and fungibility for public cryptocurrencies which have every transaction published and downloadable by everyone, and this is very bad for privacy and for commercial confidentiality especially if your amounts are on the blockchain. This makes it difficult to do business, when you're revealing all of your financial transactions, and this compromises any sort of real commercial use of this system. Using scriptless scripts, we avoid revealing contracts because all that hits the chain are public keys and signatures.

Schnorr signatures support scriptless scripts

Let me get into some algebra to explain how this works. Probably most people here are familiar with Schnorr signatures or they at least saw them in school or something. Let me briefly overview how Schnorr multi-signatures work where you have multiple parties and you want to create a signature that every participant needs to contribute to produce the signatures. They all have their separate public keys. They sum these up to get a joint public key P and they want to produce a signature which validates with the key P such that all of them together would need to produce this. So they do the standard Schnorr signature thing which is that they think of a nonce R which is actually k * G and you produce the signature s = k + ex where k is your secret nonce and e is the hash of the data going into the signature. To do a multisignature you just sum everythin so everyone chooses their own R = k * G. Everybody passes around their different R values they pass them around to get a joint R value. Using the joint R value everyone computes their joint hash challenge and then they do the same thing except e is now a hash of their joint public key and something else. Your nonces are the sum of everyone's contributed nonces, and the signature is the sum of everyone's contributed signature values. Very easy. In practice, I should warn people that there are things called key cancelation attacks where people can choose their keys and nonces in adversarial ways and you need to be careful about it. But it's not an impossible problem and it wont derail anything that I'm talking about here.

Kind of a philosophical point is that these multisignatures are already kind of a scriptless script in the sense that you have a bunch of people who have all of their own independent public keys and they add them together to get a joint key. They have a joint key and joint signatures. Public verifiers that weren't party to that, won't know how many people were involved, or that there were more than one person involved. They certainly don't know the original values. You can generalize this. If you look in the literature you'll find threshold signatures, a generalization of this to m-of-n signatures using linear secret sharing and this nice property that because a multisignature came from adding up everyone's nonces and signature values, then if you put a linear secret sharing scheme on there then you can basically do the same thing where you're contributing shares of signatures instead of entire signatures and it all just sort of works- magically.

Adaptor signatures

Alright, so, I'm going to modify this multisignature protocol a little bit to produce an "adaptor signature" which will be a building block for all of the scriptless scripts stuff that I've been working on.

What we do here is that one pary, let's just do the 2-party case, it's much simpler for 2-parties. Somebody rather than giving their nonce R in the multisignature protocol instead they think of this blinding factor T and they send R + T instead. They do everything else the same way. They are also going to send T. This is kind of a weird thing to do because I'm not trying to hide the nonce or blinding it, all I'm doingg is offsetting it with some value that I know. So we do all of this, we follow the protocol in the same way that I described it. You will get a signature that is almost valid- it's valid if you offset it by some secret value little t that only one party knows. And it's easy for everyone involved in this to verify that this is happening correctly. So what you have is basically an adaptor signature where knowledge of-- if you know the adaptor signature, the knowledge of a valid signature with the same nonce (enforced by the hash being the same), is equivalent to knowledge of the value of little t here. This is a building block.

Features of adaptor signatures

We can do a lot of cool stuff with this. It's really general. When I said big T I just threw a point over the wall and that will work and will get you this system. But if you attach any other auxiliary proofs to T, or if you're doing some other complex multi-party protocol in parallel and this big T is something important to the protocol, then what you got now is an adpator signature that will let you translate correct movement of the auxiliary protocol into a valid signature. In particular, if you're doing blockchain stuff, then you can create transactions that can only be completed by correct execution of this auxiliary protocol. And this is extremely cheap to do.

That's really exciting. It means that some semi-honest systems where there are incentive problems and you have to add a lot of auxiliary proofs, maybe you can just stick a monetary incentive for people to keep working, like say I'm doing a multi-party protocol with someone and I put up coins that they can take and I first put them into a 2-of-2 multisig output with myself and my counterparty, I do the adaptor signature protocol such that the only way they can complete the protocol is by revealing this secret value which I then need for other reasons.

What I really want to emphasize to this audience is that this is a very general framework for attaching valid signatures to arbitrary discrete log based protocols or any protocols where your primitives are linear. There's a lot of open surface area for exploring this concept.

Example: Atomic cross-chain swaps

Let me give one example of this. And then I'll close for questions. Which is the atomic swap. This is your standard proto-typical smart contract that you think about when you're doing blockchain stuff. You have Alice and Bob, and Alice has A-coins on the A-coin blockchain and Bob has B-coins on the B coin blockchain and they want to exchange these. These two blockchains don't know about each other and they can't verify each other's transactions. But somehow they want to make these two transactions happen atomically on both chains: either they both go through or neither of them do, so that there's no risk that one counterparty steals the coins and the other is left trying to be honest.

There's a classical way to do this in blockchains where you use the blockchain script system to put a hash preimage challenge and you have to reveal the same preimage on both sides. Alice knows the preimage, and reveals it on her side, and then Bob copies it on to the other chain and takes his coins. This is how the atomicity works. Here, using adaptor signatures we can do something simpler where both Alice and Bob put up their coins into 2-of-2 multisig outputs on both blockchains and then Bob gives Alice adaptor signatures using each side using the same T value which means that for Bob to take his coins he needs to reveal T and for Alice to take her coins she will need to reveal T and Bob actually knows T. So they do this setup and then there's some ordering constraints which I'm not going to go into; but they do the whole setup, and in the end nobody has an opportunity to take their coins without everything being setup such that when Bob takes his coins he publishes little t and Alice can compute little t from the final signature that hits the blockchain and uses that to make another signature that gives her her own coins and you get atomicity this way. You still have this exchange-of-information but you don't have explicit hashes or preimages on the blockchain. Nobody can even see that this was happening; and the system is much smaller, and in a public validation sense, all that anyone would need to verify is that the signatures are valid. So that's pretty cool. As a bonus you get this privacy and stuff.

Open problems

I have a list of open problems. The main one that I'm thinking about most this morning is quantum resistance. Everything I have described uses these really simple linear properties of digital signatures. I think a lot of post-quantum schemes out there like NTRU and friends are using lattices which are also really linear objects. Maybe there's a way to do this in a quantum-hard setting which would be really exciting because as much as I like cyclic groups and discrete logarithms unfortunately it seems that their days may be numbered and I'll have to start doing some real math.

That's all I have. Thank you.

Q&A

https://www.youtube.com/watch?v=ovCBT1gyk9c&t=27m47s

Q: I have a question. The blinding factor R becomes my private key?

A: Yep.

Q: How do I prevent other people from coming up with the same R value?

A: R is a uniformly random point in the scalar group. It's a 256-bit number. Assuming that the discrete logarithm problem is hard, then nobody can compute R, for the same reason that nobody can compute a secret key from a public key. This is exactly the same security model here.

Q: Also I was thinking about, because of pedersen commitments, if I know the discrete log between the group generator like G and H,...

A: That's a really good question.

Q: If I know G is xH....

A: Something I skipped over, if someone knows the discrete log between G and H, then it's not a pedersen commitment but a chameleon hash. We need G and H to be uniformly random points. So usually you have a hash to point function to get uniformly random points that come out of a random oracle. And then assuming the discrete logarithm problem is hard and assuming that you actually have a random oracle, then nobody knows this discrete log and nobody can compromise these pedersen commitments.

Signature aggregation (2017)

http://diyhpl.us/wiki/transcripts/bitcoin-core-dev-tech/2017-09-06-signature-aggregation/

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

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.

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.

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 $sG = R - H(R \| P1 \| C \| m)P1 - H(R \| P2 \| C \| m)P2$. *Yes. So I've taken the signature equation, and you get this ...* $R = H(R \| P1 \| C \| m)P1 + \ldots + s*G$. 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.

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.

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

BLS signatures for bitcoin: A conversation with Dan Boneh (2016)

http://diyhpl.us/wiki/transcripts/2016-july-bitcoin-developers-miners-meeting/dan-boneh/

https://bitcointalk.org/index.php?topic=1377298.0

Let's talk crypto. One of the things we have been doing, one of my students Alex has been talking with Greg and others. The question is:we would like to suggest, we're just doing research who knows if this catches on; a more appropriate signature for bitcoin. ECDSA was a poor choice. If you move to Schnorr sigs, which you should, there are other signatures that have other interesting properties. I would like to tell you about signature aggregation and such. Is that interesting and appropriate? Let me explain how this works.

BLS signatures for bitcoin. Me, lin and shocakum. So the point of these sigs, we were interested in them because they were short, but they have other interesting properties for bitcoin. Let me explain how they work and how the properties operate.

The setup is that there is a, .. i'm not sure what I can assume in background knowledge. Ican stop and explain if necessary.

There is a group, G, of order p with a generator g in G. normally you would use this group, say zero to p-1 mod p, all those numbers up to this, ... DSA would use the group zero to p-1, and ECDSA uses the elliptic curve group. Here we are going to be using an elliptic curve groups, but not the ones that NIST suggests. We are using pairing friendly curves. They have been around for 17 years, there are commercial systems using them. They are deploying them in lots of places. At (mumble) anything you buy is signed by these BLS signatures. Home Depot uses them. They are widely deployed. I can talk about why they are using them, but not right now.

I want to talk about the property of these signatures and see how they apply. The magical property of this group is in fact that, they have what is called a pairing. Iam going to write it like this. Given an integer x, in the range 0 to p-1, I can obviously compute the generator g to the power of X a member of the element group G. I can do exponentiation in this group. This is an elliptic curve group, but not a regular one. It's a pairing group.

e is takes two group elements and maps it to some other group like G prie. If you want to be concrete, every element in this group is an elliptic curve group and we can take two points and map them to some other group. The property is that this pairing is something that is called, it's what's called bilinear. We will write out the property and show how it is useful. If I compute a pairing e(gx, gy), what you get is the pairing of... G to the power of X times Y. This is the fundamental relation that makes this pairing interesting.

I give you gx and gy. All of the sudden the x and y comes out of the parentheses. Is g prime of the same order p?It's the same order as p. Exactly.

Just using this one magical property, we can already build amazing things. Let me show you the simplest thing. That's BLS signature scheme that works as follows. The setup is that, in key generation, it's really simple. All you do is chose a random number, i will use alpha to denote secret keys. You choose a random number between 1 and p-1. This will be your secret key. The public key is just g to the alpha. This is the same as ECDSA. You chose a base, you raise the base to the exponent.

Signing here is a lot simpler. If I want to use my secret key to sign a message m, all Ihave to do is just compute the signature, which I denote by sigma, all I do is compute a hash of the message. The hash goes from .....you take any message and map into the group g, done efficiently. We raise it to the power of alpha. That's the whole signature. It's much simpler than Schnorr and ECDSA. You take a message, map it to the group, and raise it to the power of alpha. In elliptic curve language, you multiply it. We are more used to additive. But please, continue. It may be confusing. Every time I say raise to the power of exponent, in elliptic curves, it's multiply by alpha.

How to verify the signatures? Signing is fast. How to verify? Now we have the public key, which is galpha. We have a signature which is H(m)alpha. So the way we verify, all we do is compute the pairing e of the public key and g, and we ask, is that equal to the pairing of the message e of hash of the message and sigma. So that's the whole verification condition.

This is much simpler than ECDSA and Schnorr. In Schnorr, it's a flurry of equations. ECDSA is definitely worse.

Here it is simple. If the sig is valid, this is the pairing of galpha and g. Oh wait I got this wrong. You swapped them a little bit.

e(PK, H(m)) =?e(g, sigma).

That's the verification question.

e(galpha, H(m)) = e(g, H(m))alpha the alpha came out of the parentheses.

On the right side, e(g, H(m)alpha). The alpha comes out on the left hand side on the left, but here it comes out on the right hand side, so we have equality here as well. If the sigs are valid, it's because of these chains of equality. So very simple signature.

You can prove that under standard computation hellman assumption, assuming the hash function is a random oracle, you can... schnorr signatures also assume random oracle. Same assumptions as Schnorr sigs, which work with general elliptic curves, and here we have to use pairing friendly crypto curves.

The signature is only one element in g. Whereas Schnorr is how many elements? It's two elements. So this can be 1/2 the size of a Schnorr signature. It's shorter. It means you can, you dont have to store as much space on signatures instead of blocks. Only a factor of 2.

That can save lot of space. Blockchain is storing lots of extra data. With segwit, different story, yeah.

Why is this relevant to bitcoin? Well there's another property, the aggregation property. Aggregate signatures. This might seem like a magical crypto primitive.

We have a bunch of public keys, like PK1, m1, and sig1. Pk2, m2, sig2. We have that up to a million public keys, whatever. Sure.

You should think about this as one block. A block has what, a few thousand transactions. About 2000 transactions in a block. About 2000 messages signed by different people producing different signatures. You have to store all these things in the block. Ignoring segwit, you have to store all the signatures in the block.

The interesting thing about BLS is that you can take all these signatures and aggregate them all together, so that you get one signature, a single group element, 32 bytes for the entire block. You don't have to store signature sat all. No signatures in the block. Well, one signature for the entire block. So, exactly, right. There are some issues with making this work. Let's talk about the math.

How does this work? Aggregation is simple. Sigma is just, you just take all these signatures and multiply them together. Each signature is just a group element. So, in the language of elliptic curves, every signature is a point on the curve. Take all the signatures and add them together.

There is by the way some technicality in what needs to be hashed, if you do it this way, you can't just hash the message, you should also hash the public key as you're signing. This is good practice in general. It's unfortunate that bitcoin doesn't do it; well they do it indirectly by hashing the txid. Generally, whenever you implement signatures... segwit fixes this. Few schemes commit to the pubkey. Commercially used?It's not typically done. This is not just bitcoin's fault. When you build signatures in the real world, it's important that when you sign things, you prepend what you're signing to the message. Otherwise you get DKS attacks. It's important to also stick the pubkey into the hash. When you do sig aggregation, it's crucial you stick the pubkey into the hash or else things could go wrong.

How do we verify that single signature for an entire block?What do you need to verify?What you need to verify is all the public keys and all the messages. Pk1 and m1, pk2 and m2, etc. And then the single signature. We are not saving on messages, we are not saving on public keys, only saving on signature size.

How do we verify it? Well, we are going to compute e(PK1, H(m1)) and we are going to multiply that by e(PKn, H(mn)) and we will test if that is equal in its entirety to e(g, sigma). That is the question to check.

Why does this work? E(pk1, H(m1) is just... ... e(gsigma, H(m1)) = ... the property of the pairing is that the alphas come out. So the alphas come out, e(gsigma, H(m1))alpha. So just like they came out, they can come back in. SO it's e(gsigma, H(m1)alpha1) * all the others of those.

It turns out this is equal to ... I can take the product and stick it on the right hand side. The product of i from 1 to n, is equal to ..r. this is just a property of pairings. If I have a product of pairings where on the left hand side everything is fixed, and on the right hand side I'm pairing with different things, then the product just moves inside. I'm comparing g with a bunch of different things, and taking the product. The gi s always the same. I can move it inside the pairing. It's the product of all of these to alpha i. Look at how we compute the sigma. So we have equality here. That's it.

What I want you to remember is that you can take the remarkable thing here, we can take signatures that were generated by different public keys under different messages and aggregate all of them. Schnorr has some aggregation properties, but they are weak aggregation properties. If 1 million people all sign the same message, then we can aggregate these, these are kind multi-signatures. Oh and it's interactive. These are called multi-signatures, you can aggregate in Schnorr if they are the same message. In BLS sigs, you can do aggregation offline.

The reason why we came up with BLS in 2001 was the original application was certificate chains. If you think about cert chains, you have 4 or 5, Google sends you 4 certs every time you go to Google. Each one has its own signature on it. With signature aggregation, you can take those and compress it into one, and anyone would be able to do the aggregation. It doesn't have to be Google. They can look at all the signatures that the CA gave, and you can aggregate it and give it out to all the browsers. It's just adding points together. Aggregation is really simple.

Pairing is a bit harder. Let's talk about timing. It's the one issue that should be discussed. To verify an aggregate signature, you realize that what you have to do is compute a pairing over here. But you have to compute a product of pairings. So say 4000 signatures per block, and 2000 transactions per block. So you have to verify 4000 signatures. So you will be doing 4000 pairing operations to do this. There are many pairing libraries out there, like 5 or 6, and they are available. Most of the pairing libraries don't support. ... the way they would do the product of pairings is the dumb way, do it separately and then multiply. You can compute a product of pairings in a faster way than normal. Whenever you compute a pairing, one of the expensive operations at the end is a big exponentiation. There is a cleanup step using exponentiation. Every time you compute a pairing, more than half of time is spent on this final exponentiation. So if you need to compute a product of pairing, it would be idiotic to do this and do exponentiation each time. Do a product of uncleaned pairings, and then do exponentiation on the whole thing at once. This is an example of an optimization for computing products of pairings.

Alex is working on trying to optimize that. He built all this. The library, the basic library just from an off the shelf pairing library is already available. BLS is now easy enough to use, now the poor guy is digging into the implementation and is optimizing and working to optimize multiplication. We wil have a fast multiply there.

The challenge here is that a single ordinary pairing operation by itself even with fast library takes like 0.5 millisecond. It's much smaller than a ECDSAsig validation. It's 7x slower than ECDSA.. This is before optimization.

Is addition of points pretty much the same?Yes the same as elliptic curve. The curve is different, but it's basically p256. Actually many of these things are j invariant zero curves that have the same speed of optimization as bitcoin secp curve has. I can use those words here?

So that's the benefit. What I'm hoping to do, I am hoping we can... Alex needs to finish the optimizations. Then we need to write a paper to explain the performance benefits. It's not all rosy. There are downsides, like if the signature doesn't verify, then you can't tell which signature caused it to not verify. SO if there is a failure, you can't tell who's to blame. This is not an issue in block verification, we only care if the whole thing is valid or not.

This part can be transferred for fpga compilation and ASICs. Yes dramatic speedup. There is a group in Japan that did FPGA implementation, they got from 500 microseconds to a few microseconds. It will be 100s of nanoseconds. The greater implementation challenge is that this implemented like you describe interacts poorly with caching. In bitcoin block propagation, we validate blocks quickly because the signatures have already been validated and cached. So figuring out how to work this with caching is a challenge. There could be ways to do caching, but they break the product of pairing optimization. So Pieter and I worked with Alex to try to figure out engineering implications of Bitcoin and some of the tradeoffs and contours and gave him some advice on how to measure this and justify it and argue for it.

In terms of how this could actually end up in bitcoin, we're far away from that. But there are incremental steps towards that the we can take. Something I hope to be proposing in a while is that we switch to Schnorr signatures as a first step. Then do aggregation, but only at a transaction level, not at the whole block level. Where you could, assuming you have many inputs, like coinjoin. No, just normal transactions. Coinjoin is usually like 40 sigs or so. Ordinary transactions often have 2-3 inputs. So we actually computed against the current blockchain, if all transactions were to aggregate, just the inputs in transactions, it's like a 30% savings on block size. So that's good, but not anywhere near aggregation sig benefits. TO deploy this in bitcoin, if we were to deploy it, it would be a soft-fork. This would be a soft-fork for BLS and transaction signature aggregation, and that's a good step to block-level aggregation. So it's a good first step that gets us there.

How validation currently works is strictly a per-input thing. And there are changes needed to have part of the validation go beyond that. It's not a problem. But yeah needs work. You don't need to aggregate all the signatures in a block, perhaps only those that you care about. You could aggregate in sets of 100 if you want. You don't have to aggregate everything together If you abandon product of pairing optimizations, you could merkleize; generate the intermediate products of pairs of signature, merkleize them, and send partial proofs from this. But this is going to be inefficient. The groups could be quite large. They don't have to be just pairs. If you have 4000 signatures... sure. But the point is that if you have all of these signatures and if you compute them and you were to take a pair and multiply them together, a tree, then I could, if you were interested in all the signatures in any diaetic segment of it, I just show you the rash tree down to this, and I show you some signatures, and here's some product of the other side of the tree that Ihave not shown you, but I have to commit to that at least. You have a product at the top that you commit to, to verify you agree, this allows you to show any sig with a log size proof, but this requires sending gt for the products which is usually big, but it's a cute trick, so it might not be worth doing. You don't have to aggregate everything, you could choose the number of signatures you're aggregating. The merkelizing trick is a good possibility.

I don't know if bitcoin will move in this direction. Quantum computing destroys a lot of things. It's a wonderful problem to look at. We have signatures based on.. but there's problems with... post quantum crypto doesn't do aggregation?No, that's a problem I am interesting in.

These are super-singular. There are multiple families. One of them is super-singular. You could go that way. You can shave off a few... the aggregate signature with super-singular curves will be bigger; there are other families of curves that will shrink this.

I wanted you to be aware of the mechanism so that when you design the next system, you know that it exists. Yesterday I mentioned that the amount of new tech that we can deploy in bitcoin to make it more scalable and expand to more use cases, I was talking about some of the things discussed here today. There are many interesting pieces of technology that, sometimes invented a decade ago, and they didn't find much of a home, that are very useful for us.

These are not NIST curves. You are using a curve, a different type of curve. It's regular elliptic curve like in ECDSA and Schnorr. They are different types of curves. Same mathematics. These curves have been around 17 years. The math behind the pairings is beautiful. It's addictive. There are many things you can do with this problem. Most problems are easy to solve with pairing. A lot of people have looked at pairing. So far there are no attacks to this that don't also apply to general elliptic curves. There are some bias in the crypto community that arises because the pairing operation was originally used as an attack to break discrete log security in some poorly-chosen elliptic curve groups. But what Dan's work done is takes this attack, and using careful group parameter choices, turns this into something productive. It's a feature, not a bug. It turns the attack into a feature. For many people, particularly more CS people, they think oh there was an attack and we're less comfortable with this. It has been standardized, 1353133 standardized pairing crypto. The bitcoin curve is not a NIST curve either.

So commercially, there's a product that secures a connection, it was convenient to use identity-based encryption, another appication of pairing. The PoS just needs to know the name of the credit card processor, not the actual key. The post office has thousands of PoS terminals. Traditionally it's quite difficult for large merchants to install a key in each one of the point of sale terminals. So that was the ease of use. Also financial applications.

The new curve type for pairing? Could you do an ECDSA operation using this other g?Let's say you want to do schnorr signatures on the same curve. Does that work?Yes. Discrete log is hard on these curves. You can use this for Schnorr. There is another problem called decisional diffie-hellman problem, which is practical on these curves. You can have schemes that are secure on secp256k1 which fail on others. For example, with some kinds of HD wallet schemes, you could see if two separate pubkeys are linked, if you were using a pairing friendly group. That's a good example. Generally the discrete log and computational diffie hellman problem, for Schnorr, are computationally hard. So yes you could use these for that as well.

Non-interactive three-way diffie hellman. But not more than three-way. Do you have applications for that? I just think it's cool. No. It's very easy to come up with cool things you could do with pairing. There are many interesting problems I have looked at in bitcoin where I say I can't solve it, well then look at pairing, and then I have figured out how to do that, then I get rid of pairing and see if the solution still works. That line of thinking has been helpful.

....

We kind of have a hash-based signature aggregation scheme. It's not really aggregation. Imagine a hash-based signature, you could use the block hash as a cut-n-choose selection for a subset of a larger signature because for a hash-based signature you can drop security by cutting off parts of the signature. You commit to the whole of the signature, you use the block hash to select some subset of the signature that you reveal. You can reveal less and less of the signatures as they get back... I see what you are saying. The argument would be that the computation to reproduce all the blocks to rig the random selection to get your random subset of signatures, is a hardness assumption for the signatures. You could kind of compress. Tailor it to the particular blockchain application. By using the fact that we have this random beacon that is computationally hard in bitcoin, we could use that to take a bunch of signatures and then throw out parts of them, which is an interesting combination. This is the best I have come up with.

Yes that's a great question, we would love to work on that. I would have deployed hashes for hash-based signatures for bitcoin years ago, if it were not for the efficiency problems. Would core developers consider moving to hash-based functions?We need to make it an option. We have people with thousands of bitcoin in a cold wallet that they very rarely move. Even if it takes 40 kb, we should secure those with a hash-based signature.

One of the problem with hash-based signature is the reuse problem. Not reusing. Maybe inside MAST or something. You can make a tree of keys, there is ultimate convergence of these schemes. We call these nonce-reuse based hash resistance signatures. You can use hash-based signatures that are only supposed to be used once, but if you reuse them once then it's okay. There's a limit. There's spinix scheme, sphinx or whatever. 40 kilobytes per signature. That's why I said 40 kilobytes per signature, yeah. Yeah we are interested in somewhat reusable, like strong claims like 2 or 3 times is okay.

So it's interesting. The structure we have is that you're supposed to use them only once, if you happen to use them twice, then the attacker would be able to forge some messages, but very few. If you use them three times, then the set of messages the attacker can forge, grows, and if you use them more, then they can forge whatever they want. This is also true for Lamport signatures. We have a slightly different structure. We should think about applying that to bitcoin.

Signature aggregation (2016)

BLS can be very interesting and compelling at the same time. The most compelling argument is the weakness in the pairing. Signature aggregation and near-term extensions, for version script. Things like key trees and poly trees. Things are complicated by the fact that this means different things.

The simplest is essentially native multisig, which is the simplest form of signature aggregation, multiple people want to sign and the result is one signature rather than multiple signatures. The second form is cross transaction aggregation, the idea that a transaction would have in the basic case just a single signature left even if there are multiple inputs and all the inputs signatures are aggregated into a single one. The third is cross block signature aggregation, and this requires BLS.

BLS is Boneh Linnschlotem... a signature scheme that relies on different assumptions than just ECC. This would allow us to go to the point where a block just has one signature left. The fourth type is OWAS where you get to the point where not only has a block just a single signature left, but you also can't see which input and which output is correlated. The way Greg described this is that your transactions would just be inputs or just be outputs.

I think there's another axis which is whether they are interactive or non-interactive. In OWAS, you can create transactions with just spending existing inputs existing coins, or just creating outputs. These are two separate transactions and you can use sig aggregation to add together after which they can't be taken apart, and then you can aggregate more pairs together and you can't see correspondence between input sets and output sets. You have a bag of inputs and output transactions and you can't tell which one is from the other.

This will lead to the point where you could say the p2p network is constructing the block. We were talking about signature aggregation. Yesterday someone mentioned block-level aggregation, yeah you can see this as p2p network trying to construct the block and miners just pick the block. Aggregating transactions is now the same thing as constructing a block aside from PoW. No more distinction between transactions and blocks. The data structure would become the same, a bunch of inputs and a bunch of outputs.

You would want to remember the smallest possible parts, so that you can track low fee paying parts. There are various complications like if you have partially overlapping relay, you give me 1 and 2 transactions given into one signature, with 2 and 3 I can't put these together anymore, you can subtract out but only if you have seen it separately. What's the time complexity? Well let's talk about the advantages first. I feel like you just answered my question.

The other interesting advantage of this is that it might tie into long-term incentives in the network. If someone aggregates transactions and gives you them, you can't disintegrate them, so this creates an anti-censorship property. If you stick your transaction with Wikileak's, the miners can't censor your transaction without also censoring theirs, or rather theirs without censoring yours. Censorship is possible but only if you censor everything. And then you create incentives to aggregate with certain folks. There might be some disincentives in aggregate. If someone has aggregated you with Wikileaks, and you're not getting mined, you can give your transaction to the miner and they might mine it anyway on its own. If miners take their fees by adding an output that takes the excess balance thrown into the pot, it means another miner can't reorg that out, and take all the transaction fees. This is super interesting because miners can't grind this out.... I posted this on bitcointalk about 2 years ago. If these are transactions they only learn about from the block, the miners can't separate them out, so that's the positive stuff.

There are some space advantages of all of these schemes. All of these do these. 4 (OWAS) less than 3 (Cross block). One-way aggregate signatures (OWAS). I can aggregate them but not separate them. BLS is Boneh Lin Sh... OWAS is also BLS.

Other than space advantages, none of them directly improve efficiency. The efficiency improvements--- multisig improves efficiency, so the first one does, the second one kind of does (cross transaction). The efficiency improvement of 2 could be done separately by adding batch verification.

Another improvement is privacy, each improves in different ways. Multisig schnorr signatures improve policy by hiding your policy, hides that you're using multisig. The example tomorrow is that BitAwesome releases their new wallet service which uses 3-of-7 multisig and they are the only ones that do 3-of-7 on the network, now you can automatically identify who they are, and this is sort of the problem that Ethereum takes to the extreme... You can do more things that look the same. 2-of-2 looks like 1-of-1, but we can make 3-of-7 make it look like it.

Cross txn and cross block improve privacy by making coinjoins economically advantageous. You save money by making coinjoins. This is an indirect improvement. Specifically, it's different from space in that you get more savings by joining with other people. And finally, OWAS improves privacy to an extreme because it's a non-interactive coinjoin is what it's achieving.

We can also list mining incentives and other incentives.

We should also list downsides. Software complexity. Less so for multisig, it's mostly client software complexity not consensus software complexity, there's basically no consensus rules for achieving basic multisig. Tree signatures fall under 1. For 2, I want to go into a concrete proposal (cross txn).

Loss of accountability is an issue for basic multisig. If you make multisig look like single sig, you can't tell who has signed. This is a privacy/accountability thing, and there's some hybrid schemes but you lose some of the space advantages. If you know all the pubkeys, it's still computationally indistinguishable. Tree sigs preserve accountability and this is included in the first option listed, basic multisig.

New cryptographic assumptions for 3 and 4, like cross block (BLS) and OWAS. These are both based on BLS which are pairing crypto, a generalization of elliptic curve cryptography that academics love and have been thinking about for decades but has never seen any production use with real value.

For advantages, for 3 and 4, we need to list convenience. All of this requires interaction steps between signers, and 3 and 4 actually reduce them. OWAS is an improvement over the status quo because now every node in the network can.... multisig is interactive between clients... you can do one-pass. By multisig in option 1, this is Schnorr multisig, and tree sigs. When you do signing, you have to make 2 trips through the devices. First you have to get nonces, then you sign. First you commit to making a signature, bring the information together, distribute it to everyone, make a partial sig, bring the partial sigs together, then make a signature. But now everyone just signs. I can draft a transaction, sign it, hand it to someone, and you can sign it and broadcast. This is a problem if you want to do things like NFC communication for signing because now it's tap, tap again, tap again. It's only twice. First to get the payment request, send out partial commitment, you can give a nonce for free if you have a good random number generator so it could be two-steps for NFC.

Another disadvantage is computational performance. Pairing operation which is required for each verification in these schemes, every signature you verify needs a pairing operation, and they cost 500 msec each, this is an order of 10x slower than our current signature verification. There's still various verifications being performed, even if there's one signature for the block.... in 1, you don't have this, now you make 3-of-3 look like 1-of-1 there's only a single verification. In the aggregate case it's O n, in the case of pairing the O n is 500 microseconds with number of messages. It's parallelizable, and it's patchable. OWAS is one signature but the computation is still O n, it's number in the pubkeys. That composition is very fast. 500 microseconds per pubkey, an order of magnitude slower than what we have now. This is an extreme computation cost. The state of pairing stuff right now is well-optimized. Boneh thinks it could be even better optimized.

People who have implemented BLS stuff, they have on-the-fly assembly generation and it generates it, figures out your cache size, writes assembly code and compiles it, I guess this was the only way to get it to have any performance that is acceptable down to 500 microseconds. It's a pretty big slug. It's cacheable. As you see parts of an aggregate come in, when a block comes in that uses all of them, you can combine the cache data and do extra work for the missing stuff. The data you have to cache is a 3 kilobit number per signature. Your cache would be enormous, to have a cache entry for every signature in the mempool. Your mempool is going to be much bigger, unless you have a 100% hit-rate. your eviction policy for the cache could be weighted by fee or something. This is not a non-starter, but it's a challenge. In Core, we currently have hit rates at like 90%, the cache is pretty big already and we are space limited and we only have 32 bytes per entry. Increasing that to 3 kilobits, that's a little blah....

The operation to combine your cached entries takes some time, but it's maybe at the speed of sha256 or something, it's Fp addition. And you have to keep up with the network. If you were looking at using this stuff in the network today on the current hardware, like on an ARM device, the answer would be no and t would have trouble keeping up. On a desktop, sure it might keep up. There's some academics focusing on a paper for BLS and bitcoin-like systems. They basically talk about their ratio of CPU time to bandwidth this would have to be for this to be a win. If the ratios are right, then BLS could be a win.

Could this be accelerated on FGPA and hardware support? Probably an ASIC dunno about an FPGA necessarily. But yeah an ASIC. And keep in mind that cross block BLS and OWAS relies on new crypto assumptions. I don't think we have a coherent way of thinking about introducing new crypto assumptions into the public network. People could choose whether to use this stuff... So, zerocash will be heavily dependent on these assumptions and many more. This could be deployed as a sidechain and whatever else.

This is opt-in at signature time... you're pre-committing with your pubkey. It's a different pubkey here. You need to use a different group, a pairing-compatible group with your pubkey. You are stuck in BLS land. I don't know why we lost Pieter. BLS pubkeys are the same size as EC pubkeys, for equivalent belief security. BLS sigs are about half the size of

Schnorr sigs, they are 32 bytes.

Cross block BLS and OWAS are same crypto system, the difference is how you make use of it in the network. We would not implement cross block BLS, we would want to do OWAS. OWAS is where you create transactions with no outputs or transactions with no inputs. You would be aggregating them. It's a transaction format thing. One difference though is that OWAS would require stunts to do that without a hard-fork.

Implementing OWAS... well, implementing the consensus rules would be easy. You care about both inputs and outputs. You would need packaged relay. The important thing for OWAS, you could implement that without relay logic implemented. It would work just like cross block BLS. There's a way to roll it out in a straightforward way. We would probably not do cross block BLS, we would do OWAS instead. It's a transaction format question. More complex to do a soft-forkable version. To do OWAS as a soft-fork might be quite complicated. As a hard-fork it's straightforward. It's on the level of CT as a soft-fork. It's not impossible. There's probably ways to do it. CT as a soft-fork would probably be harder.

So that's the BLS stuff. I didn't talk about Schnorr stuff. With OWAS, you move the risk of losing funds to ..d.. we can't ever do things in one step except to force every wallet to send to a new address. Given a 1 megabyte size, you could have 10,000+ signature verifications that you have to do for that block. At 500 microseconds each. So that's seconds. That's why I said this is a problem on like low-end hardware like raspberry pi, they couldn't keep up with this, and on a fast desktop sure but we have to solve relay, weak blocks and caching really well. It's on the verge of viability. Maybe one more cycle of Moore's law if it holds out that long.

OWAS + CT that will be too much, pushing it too much. CT is much faster. OWAS + CT would be... well it would be zerocash but with much better scalability. Excited about the future of BLS and OWAS stuff. Someone in Boneh's group is looking into making this viable for Bitcoin-like systems. Alex L maybe.

Concrete proposal for Cross Txn, every transaction ends up with a single signature even if it has 100 inputs, even if it's coinjoin especially if it's coinjoin. This is a specific advantage. Even with a coinjoin, it would have a single signature which means you get a fee advantage from doing it with coinjoin. It's interactive. Coinjoin is also interactive. You have to do BLS stuff to get non-interactive. It's one extra round over coinjoin.

OP_CHECKSIGSCHNORR and you're done with basic multisig on that list? The thing we can leverage from Schnorr. I'll go specifically about that, and then how you apply that in 1 and 2. You can jointly construct a single transaction which is adding multiple people sign the same thing, add the sigs together, the result is now a signature that is valid for the sum of their pubkeys. If I want to send a 2-of-2 multisig to you two, you both give a public key, we add the pubkeys together, now everyone who wants to send money to them you send to the sum of the pubkeys, the only way to sign is if they both sign and add their signatures together since nobody has them. And the sender has no idea that they are using multisig. It's naive multisig, just add signatures together.

They need to jointly... there's an extra round because they have to jointly construct a nonce. Nobody actually knows the nonce you're signing with. You sum your nonces and that's the nonce for the transaction. Knowing your nonce, you form your signature and sum the signature. If you knew the nonce used for the signature, you could determine the private key, but neither of you know the nonce... in 2-of-2, why not subtract your nonce? The resulting ... these aren't nonces, they are keys. You need the sum of the pubkeys. In this case, this is the simplest case, like 2-of-2 or 3-of-3, it looks like a single sig and a single pubkey on the network. And the CPU cost is 1 verification. It's basically single verification. The network cannot tell that it is indistinguishable from a single pubkey.

Threshold sigs for ECDSA achieve the same end using a really complicated procedure in the signing which has like 6 communication rounds and requires poly N and it's really computationally expensive and requires probably 20,000 extra lines of code and a completely different crypto system.... But on the network it looks like ECDSA, people could be using that today and we wouldn't know that today, and as far as we know nobody is doing that. I wanted to do that in the past but it was a lot of code to write.

Say we wanted to do 3-of-3 schnorr multisig, get efficiency, privacy, and size advantages of this. There's a problem. The problem is say we do 2-of-2, both of you, you say my key is A, this is your actual key, and your actual key is B, but what you announce is B minus A. Now we add them together and everyone thinks to pay to you 2 the sum is A and B-A and unknowingly we just end up with B which is Mark's key and now Mark can sign for the both of them, and the other person cannot, he doesn't even know he can't. Effectively both you together are unable to sign, except by just Mark doing it... This is bad. This is annoying. This is also a problem you would have ended up if you had read academic crypto papers on this, none of which mention this problem. And usually it's not mentioned because there are external procedures that assert which key belongs to who. In theory, this is the case because you're still getting this address from somewhere, and it's something you trust and it's not unreasonable to expect that you trust it to do due diligence that both of you have your key, which is easy that you sign your own key and you're unable to do so if you have done this trick. So one way to address this is to require an interaction, pushing the complexity to the assumptions under which the scheme is valid. In bitcoin we don't like that, everyone implements their own crypto and they will get this wrong.

There is a solution that I came up with a couple months ago, which is you multiply everyone's key by the hash of their own public key. Before signing, you multiply your private key with the hash of your public key, you multiply your private key with the hash of your public key, to add our public keys together, we do this preprocessing step which doesn't otherwise influence the schnorr procedure, but we can prove that if you know his key you cannot come up with any key ... A*H(A), Mark times to come up with an X*H(X) such that X is something for which he knows K and this is impossible. If he can do this, he can break Schnorr itself. There's a black box reduction to this to making forged signatures. If you have only seen A, and if you can pick an X and a k such that this equation holds, then you can use this mechanism to break Schnorr which we figure is highly improbable, and computationally infeasible, otherwise we wouldn't be doing this. This is sufficient for the multisig case. It's not a full solution.

This does not require special interaction, the keys could be had from a database. This applies to multisig. We could do this. We can create an opcode like OP_CHECKMULTISIGSCHNORR, it takes as input different public keys that are pre-multiplied. The script logic, to do multisig, doesn't need to know about this blinding scheme, we just do it before it ends up on the blockchain. It's just OP_CEHCKSIGSCHNORR or something. It could take 1, or it could be an OP_CHECKMULTISIGSCHNORR which takes a number of public keys, then a bitmask of which one ssigned, then you have a k-of-m constraint on it. It takes the public keys, add them together, then do Schnorr verification which is really fast, which is essentially zero overhead. the point addition is basically free, it's 60x faster than signature verification. About 2 orders of magnitude.

Wasn't there a better scheme than bitmask for more complex things? That's the key tree signature stuff. It has partial overlap. In some cases, one is better. We have some other schemes that build on top of htis, like getting accountability, or poly sig which allows signing with k-of-n where k is very close to n. But let's not talk about that. For really large like 999-of-1000 and you do it in size O ten... signature is in the size of the... anyway that's out of scope here.

We implement checksigschnorr, multisigschnorr and everything up to here is as far as we know is solved. Which gets us, and it's not just perspective here, we have a schnorr signature in libsecp256k1, it's used in elements alpha, the specific formulation we're using we would like to change since we got smarter over time, but most of the work for that is basically done. We would write a paper about it before implementing it.

Now I want to talk about signature aggregation across different inputs. Here, another problem arises. This proof that we have,... well first state the problem. What are we doing? What is aggregation? So we have a UTXO set with a bunch of public keys. We have a UTXO set in the network today, there's a bunch of coins, Alice has 10 different TXouts, she would like to spend all of them at once in a single transaction and would like to do so in the simplest way possible. Even if you assume the 2 inputs belong to the same person. So we don't have the complexity of interaction, we have public key 1 p1 and public key 2 p2, which are the public keys, those are Alice's TXouts, you could write a transaction that spends those out. There's a single transaction with two inputs and two outputs. There's input1, input2, output1, output2. Only one of these would contain a signature. The way we achieve this is by applying the same technique as multisig. You gather the txouts from the blockchain, you would provide their public keys, the transaction has access to the pubkeys, you would ask the network to sum the public keys, then you would provide a signature for the sum of the public keys. It's like multisig. Then you do some layer violation breaking where the way we've been thinking about it, each of the inputs would do some setup where they say you're going to sign with that pubkey. It would be a checksig with a zero length signature in all of those. It does not interact with different sig hash. It's basically a SIGHASH_ALL in those cases.

How I would think you do it is with transaction validation, not just input validation, because we're clearly doing something at the transaction level that we were not doing before. For transaction validation, you would have an accumulator, a stack of verifications to perform. Every time you see a CHECKSIG, you would say yep it's succeeded, even in the normal script verification it succeeds automatically but as a side effect it would push the message pub key pair that it would be validating on to the stack. At the end, you look at the stack, after validation, it would have one signature and a bunch of message pubkey pairs, it would add the pubkey pairs together, add all the messages together and hash them into a single message, and verify that the sum of these public keys is a valid combined with that signature is valid sig for this [hashed?] message. It would be an OP_CHECKSIGVERIFY, or we could have an OP_CHECKSIG, it would be an assertion as to whether the checksig is supposed to pass or fail, you would pass a bool for expectation.... you have n checksigs? If you have n checksigs that are executed in a way that we assume they are valid. You have the success ones and the fail ones. The fail ones you throw around. Instead of a signature there, you provide a 0 or something, it would be ignored and execution flow would continue.

How does this interact with sighashes? Clearly, in our signature, we're now signing the hash of all messages which you can only do if you know all the messages. If you want to do SIGHASH_SINGLE, and you don't know the other inputs yet, there's no way you can sign for the other inputs. Imagine it's n inputs. Could you have one SIGHASH_SINGLE and one SIGHASH_ALL? And the ALL is the aggregated thing? A way to do this, because the scheme we've explained so far does not address this, it can only do SIGHASHALL so far. In a more generic way, sometimes you may have a part of the signers that cannot interact with some of the other signers. This is the inherent problem you are trying to address with sighash types. In this scheme, you can solve this by having multiple accumulators, the signature would contain a number saying I belong to group 1, group 2, or I belong to group 3, and for each group you would require exactly one signature, and 1 through n pubkey message pairs. There are interesting malleability ... any time you have sighash flags, you get interesting malleability issues.

If we need to implement the aggregation, it's just as hard to have multiple accumulators rather than just one, the amount of code is basically the same. However, now we come to the real problem of the linearization scheme from before, multiplying with the hash of the keys is not sufficient. There's an attack where, there's multiple ways it's insufficient. If you were doing the precomputing thing where you expect your users to delinearize their keys... well the attackers could choose not to do that. Maybe you take p1 from Alice, but p2 is from Bob which is the negative of Alice's key, so maybe it just didn't occur for Bob p2 - the delinearization didn't occur there maybe. The attacker can add an extra input, use the pubkey as a negation of the others, and then be able to spend everything. So the verifier has to do the delinearization in the network. It has to compute the hashes from the pubkeys, and it has to multiply them, to do the delinearization. That's why the performance is worse than batched schnorr validation, because the verifier has to take every pubkey, multiply it by some hash value, and some the results together.

You have 10 inputs, each of which are doing basic multisig, and then comparing that to everyone in the inputs doing cross transaction. It's faster, but not a basic multisig speed. It's in between. nominally it's the same amount of work that you are now doing per verification, even though it's a single signature. However, for free you get batch validation and this makes it faster again. Batch validation is something we could in theory do independently by why bother if we can get all of this at the same time.

The simple linearization scheme from before is actually insufficient in this case, and you may have not heard about this because we only learned about this recently. The attacker can pretend to be multiple people at once. He cannot come up with a single key which cancels out everyone else. But if he be 100s at the same time, there's a way to speed it up which is likely sufficient to break it. You have a key in the UTXO set that is you know 100 bitcoin coins or whatever, and someone goes and makes 100 1 satoshi outputs and then they are able to spend the aggregates of your coins and the 100 1 satoshi outputs, because the mixture of the pubkeys ,even with the delinearization key is still able to cancel out your keys. It would generate 100s of possible things to spend with, gazillions or such, and find some subsets of them that cancel out existing pubkeys, and this is likely enough to make the security ungood... you could do this with a feasible number of outputs, there's guaranteed to be ... guaranteed to be a solution. It's a linear combination? It's an unlinear combination because of the hashes in there. Finding it would be computationally feasible? It's logarithmic in time. Wagner's algorithm that is able to solve problems of this kind, efficiently.

You give me a 256 bit number, and you give me, and you give me 256 randomly generated other numbers. I am in log n time, so order 256 times elemental operations on these numbers, find a subset of those 256 whose sum is the first one I gave. And the way it works is you sort them all, all the numbers, you subtract two top ones, and now you have a combination of two that results in a number that is slightly smaller, and you keep doing this until all of them are 255 bits left. And now you take combinations of 255 and subtract them from each other, and I'm guaranteed to end up with combinations that are only 254 bits, and I keep doing this, and at each step I gain one bit. The list grows exponentially, but there's more efficient ways to do this without exponential blowup in space. What you would do is before creating those inputs, you would create thousands and thousands, and you would throw most of them away, and then you would put some of these inputs into the blockchain, and then you could steal the coins. So the old scheme is hash(A) times A, plus hash(B) times B, but instead you first compute S = H(A || B || C ||D) and then you add H(A||S)) and then you take.. the hash commits to all of them, now you can't have an algorithm that makes progress. When you want to add an extra input, then the whole delinearization is done again and their previous work is thrown away. It's just hashing. You cannot precompute anymore, you can't cache anymore. We believe, we don't have proof, we believe that this is secure. We have not yet got a security proof for this one. It was found while we were trying to find a security proof for the other one. Pieter was unable to find a security proof, so Adam and I said let's assume it's broken, how do we exploit it, then we found Wagner's algorithm, which is not widely known but pretty cool. It's modular subset sum.

A very specific question is, it's likely that we need something like this of this complexity for the signature aggregation scheme, but not for multisig case. In the multisig case, the delinearization scheme with pre-delinearized pubkeys is sufficient and much easier. Are we going to go to the trouble of doing both? If we do multisig with this scheme, it becomes less efficient because now the delinearization has to be done on chain rather than before it hits the chain. It's not unreasonable to do, but one costs more than the other or something, it can be within, there's just an opcode to do multiple checks and it uses the simple scheme but on top of that it uses the more complex schemes where you do it across inputs. But maybe in some cases you, the point is you can do this recursively. And now what if the keys that end up in the blockchain are the result of already some aggregation scheme that is more complicated? I would feel more comfortable if we know that the construction was the one we knew was safe. It's a very significant performance cost to the multisig.

You can use this for multisig but the problem, in the simple scheme, we can say that the pubkeys that go into the script are already pre-delinearized. We already know they are legit and conforming to that. Yes, if this was done off-chain, you can use this scheme off-chain too. But Pieter was talking about doing it on-chain for multisig, you don't have to, but it's safer that way, because that way the user didn't have the opportunity to screw it up.

We've only talked about m-of-m where all of the keys sign. Say we want to do 2-of-3. Now there are 3 different possibilities. What we can do is use this more complex scheme, do it on the 3 different subsets, have a script that says sign with this pubkey, this pubkey or this pubkey, and there's a single signature, and 3 aggregated delinearized keys on the chain. But you can use your tree signature scheme or MAST? Yes you can do that on top. This becomes infeasible when we're talking about 10-of-20, maybe not enough. I don't know the numbers exactly. For some size of number of public keys, this becomes impossible to do because it requires to create the address that you iterate over all combinations that are all of them pre-aggregated. And this easily becomes infeasible, it's not very hard to come up with cases where you need 1 year of CPU time to make the address. 3-of-2000 is still feasible, 50-of-100 is doable. But 6-of-1000 is not doable, because that's 2^64 work to just compute the address. However, if we have a native multisig opcode, where you pass some pubkeys and it adds them together at verification time, you have 1 signature, 60 pubkeys, a 60 wide bitmask with 3 bits set. And the opcode itself would do the aggregation. And now we can say, well, if this uses the simple scheme, this is super fast, still as fast as a single verification, whereas using the more complex delinearization, it needs to do EC multiplication for each of the keys and it's as slow as a batch verification of 3 signatures. For cases where tree sigs cannot be used, it would be more efficient to use a simple delinearization scheme with pubkeys that are pre-delinearized, and that is incompatible with more complex scheme which is secure in more cases. If we're talking about aiming for the best efficiency, we probably need to do both.

In case of having to do with this delinearization from every node, how much faster is it to just validate multiple signatures? So you shave off 2-3x with maybe some other optimizations. It's better. But not having to do with the delinearization in the network, is way faster. It's basically O(1). If you have something with a thousand signatures, it goes to 1 instead of 500ish. Maybe it goes to 2 or 3 or 4, whatever, for the additions. In the simple case, every additional signature adds the cost of 1/100th while in the more complex scheme it costs 1/2 or 1/3rd.

None of this is necessary in BLS. Does the Wagner attack impact polysig? No, it doesn't. You have to delinearize polysig too, but it does have to be delinearized in the network.

Polysig already pays the cost of big multiplications all the time. You can't delinearize polysig in the network because you don't have the data. Polysig is a scheme that does 9-of-10 multisig with size 1.

We have three schemes that start with Schnorr multisig, they start with Schnorr signatures and get you better multisig. They are better in different ways. The things they are better at is accountability to figure out who signed, and basically, and maybe some improvements in non-interactiveness in terms of key setup for thresholds. The 3 schemes are one that Pieter mentioned which was Schnorr check multisig, where you give the set of public keys in, and a bitmask of which ones which will be signing which also goes in, the verifier sums up the pubkeys, maybe does delinearization and then provides one signature. This is advantageous in that it's accountable, anyone can see the keys. The use case, here, is that I want to setup a honeypot, I have 1000 machines ,I want to know when one of them is broken into. I have 1 bitcoin that I don't mind losing, I can only put 1 millibit on each machine, but if I could have a 1-of-1000 multisig, I could give each machine the same 1 bitcoin, but it's only useful if I can see afterwards which machine was broken into. There's also other reasons for accountability like legal reasons. This scheme gets you accountability, but it's size O(n) in the number of pubkeys, you don't want to do the 1000 pubkey case in it. It's efficient to compute because you add up the keys especially when the number of keys is low. That's one scheme, it's obvious to do, is no brainer. The checkmultsigi thing also, if you want to do a threshold thing, like 2-of-3, each of you have pubkeys and you want to do 2-of-3, if we try to do the O(1) Schnorr, that works for combined keys n-of-n but not really well for threshold. To do a threshold, you have to do an interactive process where you end up computing differences between keys and sharing them amongst each other, it's not n squared it's worse. It's exponential in the number of... but for actual normal 10 of 20 or whatever, network protocol can do that in a couple seconds. Also it's completely insecure in the presence of key reuse. Nobody does that (laughter). We don't think people will do the O(1) things because you lose accountability, and you need complex interactive thing, and with checkmultisig scheme you get accountability, you get space savings, no interaction to setup the keys, interaction to sign, but not for 1-of-1000 because you need to put 1000 pubkeys in the chain.

The second is key tree signature. You can take n-of-m threshold and decompose it into the satisfying n-of-n key. If you have n-of-m, there is a set of n-of-ms that will satisfy n-of-n. Say you want to do a 2-of-4, there are 4 combinations, AB AC AD BC BD CD, there's 6 combinations. I can effectively turn it into 1-of-6 by expanding the combinations. Now I use this scheme of making a merkle tree out of them. My script pubkey is the root of the merkle tree. It's similar to MAST stuff. It's key-only MAST. You can do this in bitcoin script if you had a concatenator operator or a OP_CHECKMERKLEPROOF or OP_CAT. Pieter implemented OP_CAT to build and verify a merkle tree in a transaction in Elements Alpha. It requires 6 bytes in the script per level in the tree, to validate the merkle tree, plus 32 bytes in the signature to reveal the merkle path. What goes in the pubkey is the root, what goes in the sig is the merkle path connecting to the roots, plus the combined public key, and a simple checksig. It's quite efficient. It has a linear property. You get this combinatoric blowup, but you put it in a log. So the size of it is always efficient. The size is always efficient, it's just, this tree might be a billion ..... you could do it with log space but not log time. You can verify in log time. The signature is log space, but signing is ... the verify and stuff, it's 50-of-100, you could never compute the public key.

There's a third scheme, which is polysig. It's using polynomials and a polynomial matrix. I'll give a simple case of say you want to do a 9-of-10 threshold multisig. This is making productive use of the "I can cancel out your public key" thing vulnerability into something useful. Say 9 people want to sign, 10th person is not there. Let me restrict this to 5... so it's 4-of-5 in this example now. Let's call this S1, the other one is A+2B+3C+4D+5E and this is >= S2. And now what I say is ... this is a system of equations, and you're, we're putting, we've computed, we compute S1 which is the sum of the pubkeys, and S2 is the sum of the pubkeys time this linear ramp here 1 2 3 4 5... So we compute in the verifier S2 - RS7..... D cancels out, 4D minus 4D becomes 0. And we've used this cancellation to our advantage because we don't need.. anymore. Oh it's the other way around. S2-4S7.. it would be minus 3A, yeah. You tell the network, basically, which ones you're leaving out. The network computes this combination, you sign this combination that the network computes. This generalizes. You can do 1 plus 2 squared plus 3 squared, one more equation for each one you want to cancel out. As long as there are keys, which are the sums of the higher power, it's just a vander boid matrix right.

You can build a merkle tree where you put S1 in there. What would be the second level? Squared, cubed, with 2 3 4, it's the coefficients, just your indexes. This is the sum of tyour square, this is the sum of your third powers, and this is the sum of your fourth powers. I only construct this tree up to as far as I want. I allow up to 4 people to not sign, so at the 5th level I just put garbage, I invent a random number and I build this tree. This is what goes into the scriptpubkey, the roots. When everybody is present, we compute the single signature, we verify it just in S1 which is the sum of the public keys. If one person is not present, we reveal this sum allowing you to reveal the merkle path from this node up, and do a signature where two people are missing. So this has two advantages. One is that it simplifies in terms of space, and signing time, and verification time to a single signature when everyone is present. If you've built some 90-of-100 thing, but everyone is online, you just produce the O(1) signature that comes out. This random number here means that nobody in the network knows how many up to how you have allowed to not be there. The only thing you have revealed is how many people were not present at signing time. You can tell who was not there, so your accountability is perfect, even if you have leaked very little about your private information in this scheme.

There's some optimizations that you can do to make this easier. Instead of using like 1 2 3 4 squared squared squared, you can use the modular inverse of the numbers in the generation time so that the verifier doesn't have to compute the inverses in the verifier. It's fast. So this is efficient when the number of missing keys is small in absolute terms. It's inefficient if you are going to do 1-of-1000 and like 990 are missing. That's efficient under some of the other schemes, right, so we already have that. Polysigs has this cool property where if they were all available, that's efficient. In any scheme, you could do a MAST scheme where you have a checkmultisg on the left branch. It's a recursive P2SH basically (MAST). We don't have an implementation of polysig yet, and jl2012 does have an implementation of MAST.

This is like applying MAST concepts plus we're using the tree thing here plus the cancellation malicious thing turned into something useful. There are use cases where this is cool. I don't know how much it actually matters, do people really want to do 90-of-100 multisig. Maybe they want to do 8-of-10... what are the use cases for... we would use this for like, Liquid is like the sidechain transaction system where there's a federation of somewhat trusted parties that sign for the transactions and most of the time they are all online. So the only time when we would often have thresholds like 2/3rds or higher in any case, regardless of our threshold, they are all online and we would be able to sign with O(1) signatures. We're not using polysig now, we care about the size of the signatures so that we can release this on bitcoin transactions. The federated peg is where this really matters too.

Any time you have greater than majority multisig threshold with larger than usual participants, this polysig scheme is uniquely attractive. The difference of revealing 64 bytes of data versus revealing 15 pubkeys, that's a pretty big difference too. You can have 50-of-100 with polysig, it's going to be size 50, it wouldn't be much better than using a key tree. And, so one thing, you can't compute a key tree with 50... it wouldn't be any better than the CHECKMULTISIG case. If it's 50 of 100, and you only have 50 available, then you use a checkmultisig.

In theory, if your policy was something like 10-of-1000 well you don't want to do that with polysig, so when your top-level aggregation scheme is a complex delinearization, you can put anything in it, you could have all kinds of opcodes that result in some verification with a pubkey and a message and all of them get delegated to the last -- all of these schemes have various ways of determining what pubkey to sign with, or what pubkeys to sign with. And you can aggregate all of it under a delinearization scheme that makes it composability-safe. Like putting it behind MAST branches.

Anywhere you have a functionary design pattern, this would apply, like sidechain functionaries, oracles, federation of oracles, and truthcoin, most of the verification people are online and so they benefit from a polysig scheme.

With signature aggregation across txins, you would require the complex delinearization. And that same technology can, you can put various means into it for determining what to sign with. And under the assumption that it's safe for signature aggregation, it's probably safe for any other use case which is a nice fuzzy feeling. One thing that I think you didn't go into, the concrete statement about what the signature aggregation would look like, but the idea that we were thinking about was an OP_CHECKSIGAGGVERIFY where you would normally input a public key and an empty object for the signature. And potentially an id number. The group number. We could say, use bip9 again to do a new soft-fork which is Schnorr, simplest thing possible, which, replaces even, we could say it replaces the existing checksig. Probably we would do it as a separate opcode, but just think of it replacing it for now. So checksig is an operator that currently takes a pubkey and a signature, but instead here it would take a pubkey and a signature and now the signature would be an ECDSA signature plus a sighash type. But it would be turned into either a sighash type and a group number or a sighash type, group number and a schnorr signature. And per group number there would be one signature in the entire transaction that provides an actual signature ,and all the rest would be just a hash type and a group number. All would automatically succeed and return true except when you pass in an empty signature. At the end, the transaction could fail. At the end there would be a second stage, not script validation, but transaction validation that would look at these aggregators that were built, but would do the magic with delinearization and batch validation that would verify all of them. Implementation-wise, this is probably not hard. For segwit, we're essentially already adding a per-transaction validation level structure which is the sighash cache which is not currently in the pull request but we'll submit that soon. Oh yeah, tadge implemented that in his btcd implementation. It works, it verifies everything. We haven't tried mining with it. And this is essentially the same data structure for caching per-transaction sighashes, could be the same data structure where we store all the message pubkey pairs and the aggregate signatures before they can get validated. And on top of that, we could do the bitmask thing easily as a bitmask multisig opcode. But all it does is provide another mechanism of pushing things into this stack, it would not be a separate, uh, every other signature scheme added are just things that push on to this stack and in the end, uh, where the complexity comes in is at signing because we don't have a nice scheme anymore where various people provide a signature and at the end you concatenate and put into a scriptsig. You are generating way more information during the interaction round for signature, and at the end collapsing it into a single signature. The reason and combining logic would be.

For a single wallet with a bunch of utxo and different keys, it's all message passing within the program. But it's still more complex. It's more complex compared to theoretically. But the signing code in Bitcoin Core is moronic and way more complex than it could be. Maybe. We haven't tried the other way. So you need a security proof for the delinearization scheme? Yes. We need that before we deploy it. It should be a new opcode. At minimum it would be new because it would be in a new segwit script version. It's not like anyone else is going to get this imposed on them. That's not an acceptable answer, "oh well you chose to use this cracked-assed thing, what do you mean this new script version we introduced was utterly insecure, you should have verified the delinearization proof yourself" yeah so we need to publish on the delinearization scheme and get some peer review. We also should implement it too, these things can happen in parallel. I'm sure that as we implement it we will have opinions about the construction that might change the proposed design slightly. We need to implement a high-speed verifier for this, because the delinearization scheme should be fused with the verification because this will asymptotically double the performance. So we need to write a verifier that does the fused verification. Would you do a number of separate groups, do batch validation across all of them with per group delinearization, this can all be done in a single operation with enough pre-processing. It's just engineering. It will be in libsecp256k1, it will be some module in libsecp, some aggregate signature module, with non-malleable 64-byte signatures.

Non-malleable sounds good. Provably non-malleable. Fixed length is also good. No DER. We struggled on trying to come up with a security proof for this, so we probably need to have a different proof approach. We should talk with the Stanford people. Academic crypto is often BLS land and things like that. The Schnorr aggregation is kind of boring to those folks.

[libsecp256k1] This is now used in the last release of Bitcoin Core. There was quite a bit of original research both in optimizing it and in verifying it. I think we've used novel techniques for both. Essentially these are not written about anywhere except in comments in source code maybe, which is not a good situation to be in. We need to write a paper about this. We've been talking for a long time about this. There's many things to do and not a lot of time.

Pairing cryptography (Dan Boneh)

http://diyhpl.us/wiki/transcripts/simons-institute/pairing-cryptography/