From Zero (Knowledge) to Bulletproofs

March 6, 2018

1 Introduction

1.1 Audience

This document doesn't really address (at least, not well) two potential audiences:

- Experts in the field who want academic rigour
- Casual readers who want a quick skim in a few pages to get an idea

So it doesn't leave many people left I guess!

But if you are very curious about: Confidential Transactions, Bulletproofs as a way to improve them, and also the underlying technical ideas (in particular, zero knowledge proofs in general, and commitment schemes), and you have an intention to understand pretty deeply, you might find this investigation at least partly as interesting to read as I found it to construct!

Knowledge prerequisites: you're not assumed to know anything about elliptic curves, except that points add, can be expressed as scalar multiples, i.e. the standard public-private key relationship like $P = kG$, with x the scalar value which is the private key, and understand why it makes sense to say something like $aP = axG$, and that if $P - Q = H$ and $Q = rsG$ then the private key of H would be $(x - rs)$. No deeper mathematics about the *construction* of elliptic curves will be relevant here. If you've already taken the time to understand e.g. ECDSA signatures or Schnorr signatures, you'll find nothing here to trouble you.

Similar comments apply to hash functions (which play only a very minor role in most of this document).

Very basic linear algebra (let's say: how to solve a set of simultaneous linear equations; what a matrix inverse is) will be very helpful. You need to know what a vector is.

You don't need to know anything about zero knowledge proofs (no puns please!). I'll endeavour to introduce (some of) the ideas behind that phrase in stages.

If you don't know how Confidential Transactions pre-Bulletproofs work, I'd mainly suggest Greg Maxwell's main, brief write-up[5]; my own detailed description[6] of the mechanics may still be helpful in some parts, if you're new to this stuff, but goes into a lot of detail of implementation in later sections that are no longer relevant here. Thirdly, there is this[7] more compact description which could be useful, too.

1.2 Motivation

In investigating and trying to understand Bulletproofs[15], I found myself disappearing slowly down a rabbit hole of interesting ideas, blogs and eventually academic papers, that talk about a whole set of ideas based on "Zero Knowledge Proofs" and specifically, how you can construct such beasts using the Discrete Log problem (I will translate to the Elliptic Curve form here), and apply them to proving things about mathematical objects like vectors, polynomials, dot products etc.

After an initial prelude about "commitments" (Section 2) which you should definitely *not* skip unless you're already fully familiar with them, I'll focus particularly on *parts* of three papers: (1) a paper of Jens Groth from 2009 entitled "Linear Algebra with Sub-linear Zero-Knowledge Arguments"[1], (2) a paper by Bootle et al from 2016 that partly builds on those ideas: "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting"[2] (in particular its inner product argument-of-knowledge; in some way this is the main breakthrough), and finally (3) the paper on "Bulletproofs"[15] itself by Bünz et al. from 2017.

As we go through certain arguments/proofs in these papers, I'll introduce some details on how the proofs work; it's far less interesting to just see the algebraic constructions than it is try to actually **convince** yourself that they do indeed achieve the rather magical goal:

Prove that some condition holds true, without revealing anything else, but even better – do it using only a tiny amount of data!

What kind of condition are we talking about? Usually in this paper it'll be something like "Proving that you know the vector to which this commitment commits without revealing it" or "Proving that the two committed vectors have a dot product of zero" etc. Such proofs are a little puzzling without context, hence the next subsection.

1.3 Why is "proving you know vectors without revealing them" even useful?

In isolation, it's not really useful for Alice to prove to Bob that she knows, say, the vectors to which the commitments C_1, C_2, C_3 committed, if she doesn't at some later point reveal them (more about "commitments" in Section 2, if you're new to them).

But this can be part of a larger proof: for example, Alice may prove that C_4 is a commitment to a scalar value t,which is the dot product $t = \mathbf{v}_1 \cdot \mathbf{v}_2$; still without revealing any of these values. That *can* be useful, and indeed, it turns out to be the central part of the mechanics of Bulletproofs for example. The Groth 09 paper uses this proof-of-knowledge-of-committed-vectors primitive to build a whole "suite" of proofs of this type: you can use a set of vectors as an encoding of a matrix of course, so you can prove not only things like a dot product as mentioned, but also other things like a matrix product, a matrix inverse, a Hadamard product, even whacky things like that the matrix is triangular.

However, the "inner product" (annoyingly this has two other commonly used names: dot product, and scalar product; if you don't remember its definition from linear algebra, please look it up) takes centre-stage, since the paper reduces all the other linear algebra-y relations it wants to prove to that form. This

document will *not* explain this, but only (in Section 3) that first step: how to prove (technically, "argue") knowledge of a set of vectors (in zero-knowledge).

1.4 Caveat Lector

This is not a complete coverage of either Bulletproofs, nor of the earlier papers mentioned, nor of Zero Knowledge Proofs (which are just explained in outline); it's more of a patchwork coverage for the purpose of building some or most of the intuitions and machinery that will put Bulletproofs in context, and perhaps encourage you to learn more about Zero Knowledge Proofs generally.

More importantly though, this is for now just a personal investigation and is bound to contain both gross and subtle errors. Corrections will be welcome, but readers should bear this in mind.

1.5 Notation

We will be working in the elliptic curve scenario, assuming everywhere the use of a curve for which the standard generator/basepoint is G, and whose order is p(NB; this is usually N, but we avoid that because is used as a vector dimension often).

- Integers (scalars) mod will be written as plaintext lower case letters (x).
- Points on the curve will be written as upper case letters (X).
- Scalar multiplication will be written with no explicit notation (xY is the scalar multiplication of the point Y by x).
- Vectors (including vectors of points) will always be written as bolded ($\mathbf{x}, \mathbf{X}$) where not expanded out into components.
- For matrices, we'll use only the LaTeX-specific "mathbb" font: $\mathbb{X}$
- An especially cheeky piece of lazy notation: we will often deal with sums of the form:
$$a_1 G_1 + a_2 G_2 + \ldots + a_n G_n$$
, which we will (often) write specifically as an "implicit product" of two vectors: $\mathbf{aG}$. Note that this is a single curve point.

2 Commitments; homomorphic; Pedersen

(I'm going to blatantly plagiarise myself for the much of this section, from my earlier document on Confidential Transactions[6]; if you've already read that, you can skip this entirely, EXCEPT 2.3.2, 2.4, which are new here, and very important).

2.1 Homomorphism

Consider this simple mathematical fact: $a^b \times a^c = a^{b+c}$. Also, the result is equal to a^{c+b}. Notice that the addition of and works, and follows the normal addition rule ("commutativity" - you can swap them round with no effect), even after we've put all the numbers into exponents. This is an example of "homomorphism" (etymology note: "homo" here means "same" and "morphism" means a transformation/change, i.e. something stays the same under some change),

that we can use to get a blinding/encryption effect – we can do arithmetic on "encrypted" amounts (in the very vaguest sense of the term), in certain circumstances.

2.2 Commitment schemes

Cryptographic commitments are a well known powerful technique. They are usually used in a situation where you want to promise something is true before later proving it to be true, which enables a bunch of interesting types of systems to work.

Commitments are only possible because of the existence of one-way functions; you need to be able

to produce an output that doesn't, on its own, reveal the input. Cryptographic hash functions (like

SHA256) perform this role perfectly. If I want to commit to the value "2", I can send you its hash:

53c234e5e8472b6ac51c1ae1cab3fe06fad053beb8ebfd8977b010655bfdd3c3

This wouldn't be too smart though; it would mean whenever I want to send you a commitment to the

value "2", I would always be sending 53....c3 - which wouldn't hide the value very well! This is a lack of what's called "semantic security". However, it's pretty easy to address this.

Instead of committing to "2", I commit to "2"+some random data, like "2xyzabc". To put it more

generally, we decide on a format like: SHA256(secret number || 6 random characters), just for example

(the real life version will be more secure of course). Because SHA256 has the property that you can't

generate "collisions" (can't create two inputs with the same output), this still gives us the same essential security feature: once I've made the commitment, I can't change my mind on what the value is later - and you can ask me to reveal it at a later stage of a protocol/system.

Notice that by combining a hash function with an extra piece of random data, we have achieved what are known as the two key properties of any commitment scheme:

- Hiding - a commitment C does not reveal the value it commits to.
- Binding - having made the commitment $C(m)$ to m, you can't change your mind and open it as a commitment to a different message m'.

This is a counterintuitive achievement, so make sure you get it before going further - I can choose a random number that I don't reveal to you in advance and append it to my message "2", and even so,

I still can't go back on my commitment. Because the output I create, the hash value, is absolutely

impossible for me to regenerate with any other message and random number. That's the power of

cryptographic hashes (in practice, what's needed for that is called 'second preimage resistance' - you can't create a second input for an existing (input, output) pair – which is easier for a hash function to achieve than full collision

resistance, but a hash function is not considered cryptographically safe unless it has the latter).

So, a hash function provides the equipment to make commitments. However, what hash functions certainly don't have is any kind of homomorphism. There is certainly no rule like $\text{SHA256}(a) + \text{SHA256}(b) = \text{SHA256}(a+b)$. So this is no good for our purposes, because we are going to need a *homomorphic commitment scheme* for the techniques we're going to use.

2.3 A Pedersen commitment in elliptic curve form

Instead of using a hash function, we will use a point on an elliptic curve to achieve the same goal; the one way function here is scalar multiplication of curve points:

$$C = rH + aG$$

Here, C is the curve point we will use as a commitment (and give to some counterparty), a is the value we commit to (assume from now on it's a number, not a string), r is the randomness which provides hiding, G is as already mentioned the publically agreed generator of the elliptic curve, and H is another curve point, for which **nobody** knows the discrete logarithm q s.t. $H = qG$. This unknowness is vital, as we'll expand upon next.

But, crucially for the rest of this document, this new commitment scheme *does* have a homomorphism:

$$C(r_1, a_1)+C(r_2, a_2) = r_1H+a_1G+r_2H+a_2G = (r_1+r_2)H+(a_1+a_2)G = C(r_1+r_2, a_1+a_2)$$

In words: "the sum of the commitments to and is equal to a commitment to , as long as you choose the randomness for each of the first two commitments so that their sum is equal to randomness for the third."

2.3.1 NUMS-ness and binding

NUMS[8] stands for "Nothing Up My Sleeve". To achieve such a thing there are various plausible methods, but in the specific case of wanting to find an H for which nobody knows the discrete log w.r.t. G, one easy way is to use a hash function. For example, take the encoding of the point G, in binary, perhaps compressed or uncompressed form, take the SHA256 of that binary string and treat it as the x-coordinate of an elliptic curve point (assuming a curve of order $\simeq 2^2 56$). Not all 256 bit hash digests will be such x-coordinates, but about half of them are, so you can just use some simple iterative algorithm (e.g. append byte "1", "2"... after the encoding of G) to create not just one such NUMS points , but a large set of them like $H_1, H_2, \ldots, H_n$. And indeed we will make heavy use of such "sets of pre-determined curve points for which no one knows the relative discrete logs" later.

Assuming has been thus constructed, let's think about whether the **binding** property of the commitment holds true:

If I made a commitment to a single value in the form $C = rH+aG$, and I later am able to find two scalar values s, b such that $C = sH + bG$, it *proves* that *both* $s = r$ and $b = a$. More precisely it proves that *either* both of those equalities hold, *or* a "non-trivial discrete log relation" has been found, which should not be possible unless you have cracked the ECDLP (= "Elliptic Curve Discrete

Logarithm Problem"). The ECDLP (as we have already described, but without the name) basically says "given a point on the curve Q which you've never seen before, can you find its private key ("discrete log") q such that $Q = qG$?" (in some half-reasonable amount of time).

If the two previously mentioned equalities do not hold then we have $rH + aG = sH + bG$ for differing scalar values, meaning we have $H = (r - s)^{-1}(b - a)G$, i.e. we have the discrete log between H and G. This was supposed to be impossible due the NUMS construction of H, unless you have solved ECDLP. Note also that in the scenario where it is not a result of an ECDLP break, but instead someone cheated and it wasn't actually NUMS, then that cheater (the holder of the dicrete log of H w.r.t. G) can open commitments to any value he likes – in other words, the commitments are no longer binding at all.

Assuming none of this applies, the binding property allows us to construct proofs like: given that C is part of the input to a system, if I have some algorithm (even involving weirdly powerful adversaries of the type we're about to see!) that generates a formula $C = r'H + a'G$, I can safely assert $a = a'$, with the above mentioned conditions.

2.3.2 Perfect Hiding and Perfect Binding are incompatible

The Pedersen commitment is not *just* "hiding" as explained above: it has a property known as "perfect" or "unconditional" or "statistical" hiding.

This fact is sort of "complementary" to the above fact in 2.3.1 – that it's not binding, if you know the discrete log of . Consider, if I have a commitment $C = rH + aG$, there is another r' s.t. $C = r'H + a'G$ for any chosen a'; it's just that we can't *find* that r' without the ability to crack the ECDLP. But the fact that it even exists means that the commitment is a valid commitment to *any* value a, for the right r. This property is shared of course, famously, with the One Time Pad[16], which also perfectly hides the encrypted value.

However, the Pedersen commitment's binding is not perfect – it is "computational". What this means is that, much as discussed just above, in a case where the ECDLP was cracked, the binding property could fail and people could open the commitment to other values.

Ideally, we'd have a commitment scheme that was perfect in both respects – binding and hiding, but unfortunately that is **logically impossible**. Consider that if the above mentioned perfect hiding property applied (as it does for Pedersen), then it is possible for more than one pair (r, a) to be a valid opening for the commitment C, meaning it is not *perfectly* binding.

Hence the title of this subsection.

There are commitment schemes that make the opposite tradeoff – they provide perfect binding but not perfect hiding. That of ElGamal[9] is one such. However to make that tradeoff it's necessary that you don't compress – the output cannot be smaller than the input, for if it was, the mapping between them cannot be injective.

Further to this last point, Ruffing[10] has come up with a new concept "Switch Commitments" by which he proposes to at least partially solve the problem of how to have a kind of mutable commitment that switches from perfectly hiding to perfectly binding at a later date.

2.4 The Vector Pedersen Commitment

To extend to a more powerful form of the Pedersen commitment already defined, we go from:

$$C = rH + aG$$

to:

$$C = rH + (v_1G_1 + v_2G_2 + \ldots + v_nG_n) = rH + \mathbf{vG}$$

The individual G_is can be constructed using a simple algorithm of the form already mentioned (like, take a $\mathcal{H}(\text{encode}(G)||i)$ where $\mathcal{H}$ represents some hash function). The opening of this commitment would of course be the random scalar r and the components of the vector $\mathbf{v}$. I defer to elsewhere, proof that this extension preserves both the hiding and binding properties, although it's pretty obvious.

The main thing to observe right now is: this is a very powerful construction if you're interested in compactness. The vector may have an arbitrarily large number of elements, but is committed to with **one single curve point** C.

A final note (and this will be used a lot): we can extend this yet further to create commitments to 2 or even multiple vectors; we just need a set of N NUMS base curve points for each vector we're committing to, for example (2 vectors):

$$C = rH + \mathbf{vG} + \mathbf{wH}$$

Note here that the single curve point H is not part of the vector $\mathbf{H}$.

Finally, note the connection with 2.3.2 above: there we pointed out that if a commitment compresses the input, it can't be perfectly binding. Here, we are doing a huge amount of compression: there are $2N$ scalars in the above vectors, for dimension N. So commitments of this type can't be perfectly binding, whether they're using Pedersen or any other style of commitment. Hence we can assert that the methods developed here for Bulletproofs (heavily using this kind of vector commitment) can never have perfect binding.

3 A zero knowledge argument of knowledge of a set of vectors

This section will go over Appendix A from the Groth paper [1]; in so doing, we'll get a chance to start to understand the basic mechanics of doing zero knowledge proofs, so there will be some side-discussions as we go through it.

An "argument of knowledge" is a technical term distinguished from "proof of knowledge" by the idea that the proof is only computational – an adversary with enough computing power *may* be able to convince you that he knows the secret value(s), even if he doesn't.

Before starting: we will not be discussing , here, removing the interactivity from this process using Fiat-Shamir / random oracle model, as it's an extra level of complexity in the zero knowledge proof aspect we want to avoid for now. We'll make some comments on it near the end of the document, in Section 6.2.4.

So here just assume that the Verifier of the proof will interact with the Prover in real time.

Our argument of knowledge will come after we have generated a set of commitments for each of m vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$, each of the same dimension $N(\neq m)$). Explicitly:

$$
\begin{aligned}
C_1 &= r_1 H + \mathbf{x_1 G} \\
C_2 &= r_2 H + \mathbf{x_2 G} \\
&\ldots \\
C_m &= r_m H + \mathbf{x_m G}
\end{aligned}
$$

Since the commitments are (perfectly) hiding, we have not revealed these vectors by passing the commitments to the Verifier. So having at some earlier time shared these commitments $C_1, C_2, \ldots, C_m$, we will now prove/argue in zero knowledge that we know the openings of all of them.

Here's the process interactively:

P $\rightarrow$ V: C_0(a new commitment to a newly chosen random vector of dimension N)

V $\rightarrow$ P: e (a random scalar)

P $\rightarrow$ V: $(\mathbf{z}, s)$(a single vector of dimension N, and another scalar)

These last two are calculated as:

Note that the summations start at 0; this means that the sums include the extra, random commitment, indexed 0, that was created as the first step of the interaction. Note also that we are using *powers* of the random number e, i.e. literally the set of numbers $(1, e, e^2, \ldots, e^m)$. We will discuss this important detail later in 3.3.1.

In case it isn't obvious why this actually keeps the vectors $\mathbf{x}$ hidden, consider one component of $\mathbf{z}$, like for example $z_2 = x_{02} + ex_{12} + \ldots e^m x_{m2}$; the addition hides the individual values.

Having received this $(\mathbf{z}, s)$, the verifer of course needs to verify whether the proof is valid. He does the following:

$$\sum_{i=0}^{m} e^i C_i \;\; =?\; sH + \mathbf{zG}$$

3.1 Completeness: does it validate if the opening is correct?

We can see this by expanding the RHS of the above verification check:

$$
\begin{aligned}
& sH + \mathbf{zG} \\
&= \sum_{i=0}^{m} e^i \left(r_i H\right) + \sum_{i=0}^{m} e^i \mathbf{x_i G} \\
&= \sum_{i=0}^{m} e^i \left(r_i H + \mathbf{x_i G}\right) \\
&= \sum_{i=0}^{m} e^i C_i
\end{aligned}
$$

So an honest Prover does indeed convince the verifier.

3.2 Zero knowledge – does the prover reveal anything more

We deal with zero knowledgeness before soundness, because the latter is the harder proof (and indeed the most interesting part!).

To argue for zero information being revealed to the Verifier (other than the single bit of information that the Prover knows the opening of the commitments), we use this reasoning:

> If the distribution of transcripts of the conversation between Prover and Verifier, in the case where the verifier's execution environment is controlled and it is run by a notional entity called a "Simulator", and we can simulate a proof without actually having the knowledge, is the same distribution as that obtained for genuine conversations with Prover(s) who *do* know the opening of the vector commitments, it follows that the Verifier learns zero from the interaction other than the aforementioned single bit.

For more details on this basic (if rather bizarre at first) reasoning for the construction of zero knowledge proofs, refer to e.g. , [14] for a discursive introduction, and [17] for an in-depth discussion with academic rigour (hat tip to Jonas Nick for pointing me at that!). The Wikipedia page[18] gives a summary also.

There's some value in chasing up those links and spending time with them before going further, although technically you have enough to basically understand it here. Here I will only briefly mention the three key properties of any zero knowledge proof:

- Completeness – does an honest Prover succeed in convincing the Verifier
- Soundness – does the Prover actually *prove* the truth of the statement
- Zero-Knowledgeness – can we reveal that the Prover reveals nothing else than that the statement is true.

In academic coverage of this concept, there are a lot additional definitions used. A "witness" is a piece of (usually secret) data corresponding to a "statement" which the Prover possesses but does not want to reveal. Zero knowledge

comes in flavours such as "Honest Verifier Zero Knowledge" with a number of subcategories. And so on. This document is not attempting to be rigorous, and so will avoid going into details at this level. If you want to do so, again, I refer you to [17], although there are bound to be a lot of other high quality resources too.

As a preparation for using these ideas in practice, here is how the proof works for a (slightly) simpler case, that of Schnorr's identity protocol (which is basically the same as the Schnorr signature, except the interactive form, and ignoring messages). To review, the basic structure is:

Prover starts with a public key P and a corresponding private key x s.t. $P = xG$.

Prover wishes to prove in zero knowledge, that he knows x.

P $\rightarrow$ V: R (a new random curve point, but P knows k s.t. $R = kG$)

V $\rightarrow$ P: e (a random scalar)

P $\rightarrow$ V: s (which P calculated from the equation $s = k + ex$)

Note: the **transcript** referred to above, would here be: (R, e, s).

Verification works fairly trivially: verifier checks $sG =?R+eP$. Now, to prove zero knowledgeness of this construction in the above described framework:

The "Simulator", which controls the execution of the verifier, given the public key P, just as the Verifier would be, can fake a valid transcript as follows:

Choose s randomly. Then, choose e, also randomly. Finally, we only need to choose R to create a complete conversation transcript; it must be $R = sG - eP$. Then we have successfully simulated a conversation which is entirely valid: (R, e, s), without ever knowing the secret key x, and which it's easy to see is randomly distributed in the same way as a real one would be (R is a free variable).

This is a useful example to start from; it shows how, if the proof relies on causality in the interaction (you only get A after you first give me B), then since a conversation transcript doesn't intrinsically enforce that, such transcripts can (at least in this case) be faked. Another way of looking at it is that this kind of proof is **deniable** – it may be entirely valid to the interacting Verifier, but entirely meaningless (as in this case) to a third party who is shown the transcript later. And though this is not *quite* the same as zero knowledge (we also have to consider distributions), it's basically the same here.

Coming back to our vector proof of knowledge, we can see that we're in a very similar situation. The conversation transcripts look like:

$$(C_0, e, (\mathbf{z}, s))$$

which is almost the same, except that the final portion is a vector + a scalar instead of a single scalar. And so the same reasoning applies: a Simulator can fake the transcript by choosing out of order (it's only a slightly more algebraically involved issue here: you choose $(\mathbf{z}, s)$ both at random, as well as e, and you can deduce the right value of the point:

$$C_0 \;=\; (sH + \mathbf{z}\mathbf{G}) - \left(\sum_{i=1}^{m} e^{-i} C_i\right)$$

(remember, the $C_1, C_2, \ldots, C_m$ are all set in advance). It's easy to see that this will mean that the entire transcript will look valid to a third party, and it's less

obvious but certainly plausible that the statistical distribution of these transcripts will be indistinguishable from that for genuinely constructed transcripts by honest Provers; thus zero knowledgeness is proven.

3.3 Knowledge soundness – does a verifying interaction actually prove knowledge of the vectors?

This is the most interesting part, and justification here will explain a lot about why the mathematical structure is what it is.

Taking an intuitive view, it makes sense that this is the most sophisticated: if someone gives me just **one** vector and one scalar, it's more than a little surprising that this is enough to prove correct opening of a potentially large list of vectors .. e.g. suppose we had 10 vectors and $N = 20$, then there are 210 different variables embedded in the Pedersen commitments $C_1, C_2, \ldots, C_m$ - 10 random scalars and 20x10 individual vector components. We'll be proving knowledge by only providing one vector plus one scalar, so 21 numbers (but not revealing any of the *original* numbers in doing so! - see the previous section).

Proving "soundness" is somehow complementary/"opposite" to proving zero knowledge, in the following sense: the idea here is to isolate/control the operation of the Prover, as a machine, rather than isolate the verifier. If we can control the Prover's environment and by doing so get him to spit out the secret information (the "witness"), it follows that he must have it in the first place! In the real world, malware aside, the Prover is interacting and will not allow breaking of his own rules, but that fact doesn't invalidate the above reasoning.

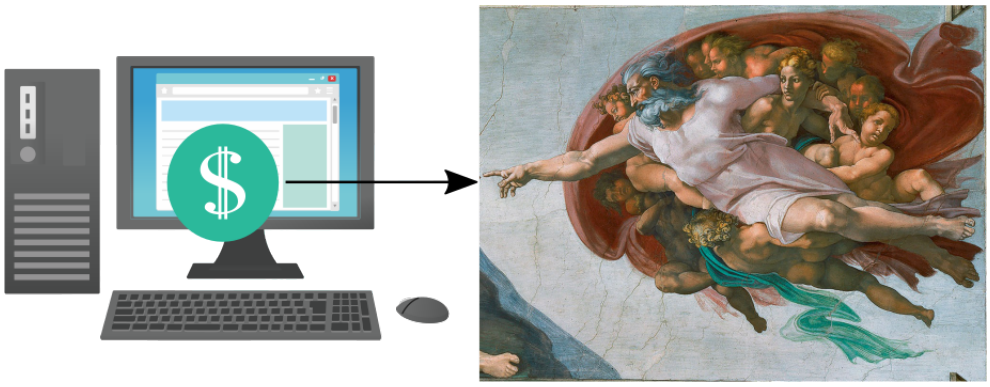

Figure 1: *God (the Extractor) stealing the secret ($) from the Prover (Machine)*

Even God cannot steal $100 from you if you don't *have* $100. Contrariwise, if you do, he can!

You might object: "*hmm, if this adversary is basically God can't he just dream up $100 for you and then steal that?*". Yes, but that's not stealing $100 *from you*. We want to know what is inside this black box machine called a Prover; it's of no interest what we can inject into it.

Another way of thinking about it, that might be especially appealing to coders: imagine the Prover is literally a function. You can start running it, stop at any point (imagine debugger breakpoints). Crucially you can make copies of its current state. You can take info from one run and feed it into another,

etc. If by doing this whacky stuff you somehow manage to get it to spit out information that reveals the secret (say, a number x such that $P = xG$), then it must have been in there in the first place.

In the Schnorr identity protocol case, this is quite straightforward: we get the Prover to run twice, but only after the same initial random point R. So imagine I as "Extractor" (what we call the "God" controlling the Prover) create two runs of the protocol with two different values e_1, e_2 against the same initial R, then:

$$\begin{aligned} s_1 &= k + e_1 x \\ s_2 &= k + e_2 x \\ \rightarrow x &= \frac{s_1 - s_2}{e_1 - e_2} \end{aligned}$$

Thus, the Extractor managed to get the secret key in two runs of the protocol that happened to share the same "nonce" point R (remember, $R = kG$ and it's the same in both runs here). This is such a widely known "exploit" of both Schnorr and ECDSA signatures (when wrongly implemented) that I won't belabour the point; but note it's crucial here to proving that the construction has "knowledge-soundness", i.e. that this protocol **can only be run by a Prover who actually knows the secret, $\mathbf{x}$**.

OK, we've given the background, now on to the nitty gritty: how do we prove knowledge soundness for our zero knowledge proof of knowledge of the openings of the commitments to the vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$?

First point:

We need to get the Prover to output not just two transcripts, but $m+1$. This will be enough to prevent the system of equations from being underdetermined, i.e. it will give us a unique solution. In more detail:

As for the Schnorr case, we have the Extractor start the Prover, who generates here a C_0, then provide it with a random challenge e, then retrieve from it a pair $(\mathbf{z}, s)$. Assuming that this response is valid, we can repeat the process, a total of times, resulting in this set of transcripts:

$$\begin{aligned} &(C_{0,1}, e_1, (\mathbf{z}_1, s_1)) \\ &(C_{0,2}, e_1, (\mathbf{z}_2, s_2)) \\ &\ldots \\ &(C_{0,m}, e_m, (\mathbf{z}_m, s_m)) \end{aligned}$$

The Extractor now effectively uses this data as a set of linear equations that it can solve to extract the values of the commited vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$. Here's how. It starts by constructing the Vandermonde matrix [19] of the challenges e_i:

$$\mathbb{A}^{-1} = \begin{pmatrix} 1 & e_0 & e_0^2 & \ldots & e_0^m \\ 1 & e_1 & e_1^2 & \ldots & e_1^m \\ 1 & e_2 & e_2^2 & \ldots & e_2^m \\ \vdots & & & & \\ 1 & e_m & e_m^2 & \ldots & e_m^m \end{pmatrix}$$

(Ignore the LHS for a moment). The Vandermonde matrix, acting on the column vector of a set of coefficients of a polynomial, outputs a new column vector which represents the evaluation of that polynomial at each of the points $(e_0, e_1, \ldots, e_m)$. What's particularly elegant here is that this means the *inverse* of that matrix, *if it exists*, therefore maps a set of $m+1$ polynomial evaluations (the polynomial here has degree m), back to its set of coefficients, and most crucially **that mapping is one-one and therefore the solution is unique.** *(This idea is used in polynomial interpolation; as you may know, a set of $N+1$ evaluations fixes a polynomial of degree N.)*

Note that this of course breaks down where there is *no* such inverse, which is easily seen to occur exactly and *only* in the case where *not all* of the $(e_0, e_1, \ldots, e_m)$ are distinct; this would represent a scenario where you had less than $N+1$ evaluations of the polynomial; the set of equations would be underdetermined. As is well known from high school level linear algebra, the inverse exists if and only if the determinant is non-zero, and this is the product of the pairwise differences of the e-values (which is a useful result about Vandermonde matrices – one for which we have just explained the insight). All we need is that the e-values are all different, which of course we can arrange to be true in this Extractor mode.

Holding in mind this guarantee that the inverse exists, you can see from the above equation that $\mathbb{A}$ is that inverse. Consider the identity:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} 1 & e_0 & e_0^2 \\ 1 & e_1 & e_1^2 \\ 1 & e_2 & e_2^2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

where we're looking at only $m = 2$, for brevity. We can see that for any particular e-value ('challenge'), the following holds: $\sum_{j=0}^{m} a_{hj} e_j^i = \delta_{hi}$; in words, the i-th row of the matrix $\mathbb{A}$ yields 1 when multiplied by the column vector of i-th powers of the challenges, and zero when multiplied by all other columns (here δ_{xy} is just a well known mathematical shorthand called the "Kronecker delta" which yields 1 when $x = y$ and zero for all other combinations).

Now we'll show in a series of steps how we can use this to extract the witness, that is to say, the actual vectors $mathbf{x}_i$. For any specific commitment in the

set $C_1, C_2, \ldots, C_m$, we'll say we're looking at C_h, we can write:

$$
\begin{aligned}
C_h &= \sum_{i=0}^{m} \delta_{hi} C_i \quad \text{by definition of Kronecker delta} \\
&= \sum_{i=0}^{m} \left(\sum_{j=0}^{m} a_{hj} e_j^i \right) C_i \quad \text{as above paragraph} \\
&= \sum_{j=0}^{m} a_{hj} \left(\sum_{i=0}^{m} e_j^i C_i \right) \quad \text{additive commutativity} \\
&= \sum_{j=0}^{m} a_{hj} \left(\sum_{i=0}^{m} e_j^i \left(r_i H + \mathbf{x}_i \mathbf{G} \right) \right) \quad \text{defn of commitment C} \\
&= \sum_{j=0}^{m} a_{hj} \left(\sum_{i=0}^{m} e_j^i r_i H \right) + \sum_{j=0}^{m} a_{hj} \left(\sum_{i=0}^{m} e_j^i \mathbf{x}_i \mathbf{G} \right) \quad \text{additive associativity} \\
&= \sum_{j=0}^{m} a_{hj} s_j H + \sum_{j=0}^{m} a_{hj} \mathbf{z}_j \mathbf{G} \quad \text{defn of s,z} \\
&\implies C_h \text{ is a commitment to } \sum_{j=0}^{m} a_{hj} \mathbf{z}_j \text{ with randomness } \sum_{j=0}^{m} a_{hj} s_j \\
&\implies \mathbf{x}_h = \sum_{j=0}^{m} a_{hj} \mathbf{z}_j \quad \text{by the binding property of the commitment}
\end{aligned}
$$

The last step of reasoning is crucial, of course; this only extracts the $\mathbf{x}$ vectors because you cannot open a commitment to two different values/vectors. See the details in the section on Pedersen commitments.

I realise it all looks a bit fancy, but that's just typical mathematical formalism/neatness; what it really means is pretty simple: if you have $m + 1$ evaluations, you can fix the mapping between $\mathbf{z}_j \leftrightarrow \mathbf{x}_j$, invert it and extract all the $\mathbf{x}$-vectors.

Through this algorithm, we were able to therefore *extract* the committed vectors $\mathbf{x}$ from the Prover, and thus we have "knowledge-soundness", that is, a Prover cannot provide a verifying proof without actually knowing the values.

3.3.1 Why do we use powers of e? Generalisations about polynomials

Referring back to the basic protocol, which is Prover sends C_0 , Verifier sends e, Prover sends back $(\mathbf{z}, s)$, we see that the Verifier only sends one random value e, which is then "expanded" into this set of powers (note btw that all scalars are mod p, where p is the order of the elliptic curve). It's intuitively logical that $m + 1$ challenge values are indeed required; imagine for example constructing $\mathbf{z} = e \sum_{j=0}^{m} \mathbf{x}_j$; this would obviously be pointless as it doesn't fix the vectors $\mathbf{x}$ at all (previously, we expressed this as the idea that a set of linear equations are "underdetermined"); one could clearly open this $\mathbf{z}$ to any number of combinations of vectors $\mathbf{x}_j$ that happened to add up to the same value. So

this makes us realise that we're going to need $m+1$ *not-predicted-in-advance-by-the-Prover* coefficients, by which he'll have to multiply his individual $\mathbf{x}$-es before adding them together. So for these coefficients, you need something like a "basis" that doesn't allow clever cancellation.

You might try to achieve that if the verifier sent $m+1$ independent random values e_j. Construction and verification of the proof would still work here, but this would waste communication cost ($m+1$ separate random scalars).

Clearly a compact transfer of only one value e is preferable from that point of view; but is this series of powers of enough to *guarantee* that the Prover's original vector (in this case $\mathbf{x}$) is fixed?

The most intuitive way to understand it: the vectors $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ can be seen as the coefficients of a polynomial $P(e) = \mathbf{x}_0 + \mathbf{x}_1 e + \mathbf{x}_2 e^2 + \ldots + \mathbf{x}_m e^m$. Note that this polynomial is vector-valued. Now this set of vectors (= this polynomial) is *fixed* by $m+1$ evaluations (see Figure 2).

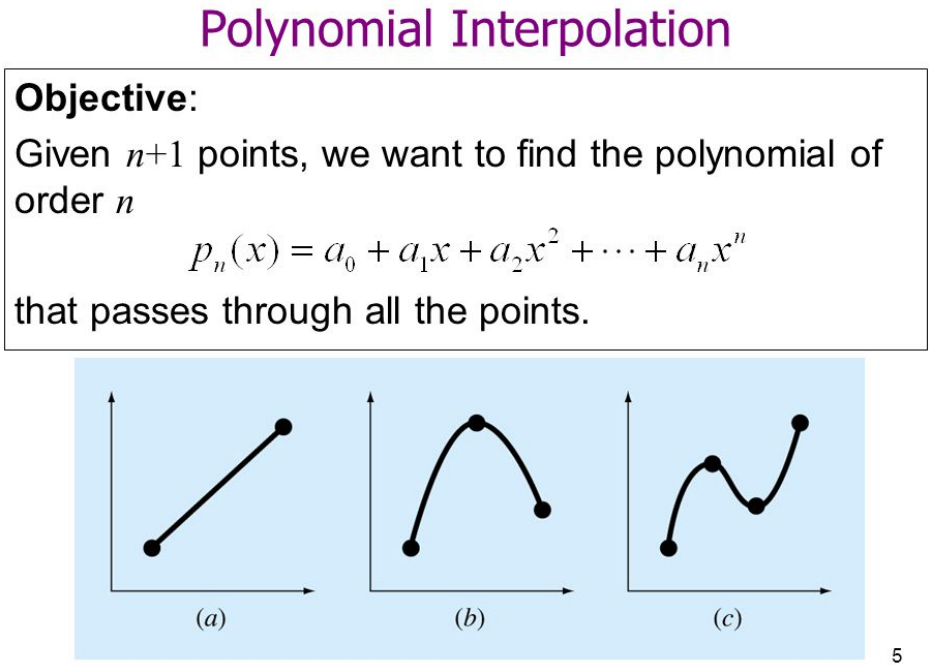

Figure 2: *A polynomial of degree n is fixed by n+1 evaluations*

It's important (for later parts of this document) to realize that one can go further here – one can assert that even a *single* polynomial evaluation of this type may be enough to fix a single vector with overwhelming probability. Consider the argument like this: imagine you have a vector $\mathbf{v}$, and you make a commitment to it C_v. Now, how can you, in a very short (low-bandwidth) interaction, prove to a Verifier that that vector was the zero vector? Let the Verifier pass a scalar challenge e, and then reply with a proof that $P(e) = \mathbf{x}_0 + \mathbf{x}_1 e + \mathbf{x}_2 e^2 + \ldots + \mathbf{x}_m e^m$, as above, is $\mathbf{0}$. Clearly without knowing the challenge in advance it's unlikely that you'd have been able to choose a $\mathbf{v}$that

wasn't **0** itself, and still have the dot product verify. But how unlikely? It's easy in this simplified case to estimate the probability: since e is randomly chosen, the chance that it is a root of $P(x)$ must be the order of (number of roots of)/number of available numbers), i.e. m/p where p is, again, the order of the elliptic curve. This is of course not a rigorous treatment, but should be good enough. For typical elliptic curves (256 bit) and typical polynomials (some countably small number of components), this will be negligible. We can use this style of argument to build commitments to a set of values, or vectors, by turning them into coefficients of a polynomial of a single scalar challenge, and have that whole set of commitments be computationally sound.

3.3.2 Aside: a philosophical musing about Zero Knowledge Proofs

(As you can guess, this section is ... not required for the rest of the document)

If you found it easy to grasp the general outline of the above arguments, then kudos I guess – for most people this stuff seems **very** weird on a first pass-through. The idea goes back to Goldwasser, Micali and Rackoff [20] from the 80s. What strikes me as interesting is that the whole basis of the idea is shifting the notion of a "proof" from complete perfection (nothing less is accepted in Pure Mathematics – axioms, syllogisms, proof, etc.) to procedures based on a fundamentally **computational** notion of soundness – the proof is either acceptable because different distributions of results are not statistically distinguishable ("statistical soundness"), or even weaker, that an invalidating counterexample cannot be constructed without computing power greater than some infeasibly large number. Of course this is totally normal in practical computing and cryptography (especially today), but in Mathematics it's of course not at all – apparently this is part of why the initial ZKP paper had to be submitted several times before it was published!

The reason I mention it is because it's of course part of the broader trend – note that the basic mathematics behind the first public key cryptosystem (RSA) was well known decades and probably centuries before the 1970s, when it was invented. It just wasn't relevant *until* computation became fast enough for it to become relevant. Factoring "large numbers" is "hard" when "large" means a few thousands or tens of thousands. That asymmetry (between making a product and decomposing it) existed, but it wasn't so big as to be interesting. But when you can work with numbers that have 200+ digits in their base-10 representation, that asymmetry has blown up to such a huge extent that you can treat it as a one-way function. So in this sense the entirety of public key cryptography is, realistically, a direct consequence of fast computation creating a kind of "phase change" in the significance of the underlying number theory.

4 An inner product proof

In Section 5 of Groth's paper, he presents the core algorithm, which probably-not-coincidentally is also the core of Bulletproofs (although the latter's version is more sophisticated and more compact, as we'll see later). The inner product proof here uses all the same elements as we've discussed above, although in a slightly more complicated structure.

It starts by assuming that the Prover has two vectors $\mathbf{x}$ and $\mathbf{y}$, and obviously knows the inner product of those, which we'll now call z.

The Prover's job will now be to convince the verifier that *Pedersen commitments* to these three quantities obey $z = \mathbf{x} \cdot \mathbf{y}$; so we assume upfront that the three commitments are known, we'll call them from now C_z, C_x, C_y:

$$C_z = tH + zG$$
$$C_x = rH + \mathbf{x}\mathbf{G}$$
$$C_y = sH + \mathbf{y}\mathbf{G}$$

(remember our notation cheat: the bolded parts are actually summations).

4.1 Aside: the Sigma protocol

This is an abstraction, worth mentioning at this point, because we are about to see another example, and we have already seen two. Here they were:

P $\rightarrow$ V: R (a new random curve point, but P knows k s.t. $R = kG$)

V $\rightarrow$ P: e (a random scalar)

P $\rightarrow$ V: s (which P calculated from the equation $s = k + ex$)

and

P $\rightarrow$ V: C_0 (a new commitment to a newly chosen random vector of dimension N)

V $\rightarrow$ P: e (a random scalar)

P $\rightarrow$ V: $(\mathbf{z}, s)$ (a single vector of dimension N, and another scalar)

(the first was Schnorr's identity protocol; the second was the proof of knowledge of a set of vectors). These are both examples of Sigma protocols, so called because of a vague resemblance to the greek letter Σ, in that the process goes forwards once, then backwards, then forwards finally. The common pattern, though, is more than this three step interactive process. We generalise it as something like:

P $\rightarrow$ V: commitment

V $\rightarrow$ P: challenge

P $\rightarrow$ V: response (proof)

It's worth emphasizing this structure, and why it arises, at this point in the discourse, so the mathematical structure of the inner product proof isn't too overwhelming. Consider the Schnorr case (basically the simplest) to see why it's necessary for the provision of a proof. Say you have a secret value x.

Option 1: just send x to V

This would be fine except we're trying to keep x secret, so ...

Option 2: send to V

This hides ("blinds" is sometimes the term used) x perfectly (in particular, to an outside observer of the communcation, so that's something), but it also leaves V with no way of verifying.

The first attempt to get out of this impasse is to let V get involved – interactivity. Hence the "challenge".

Option 3: V sends challenge e, P sends $x + e$

Dumb of course; this doesn't hide anything from V since V knows e.

At this point we would just be stuck and give up, except for one thing: we have *one-way-functions*. This means P can send a *commitment* instead of a

naked value (not a properly hiding+binding commitment, so using the term loosely):

Option 4: P sends R, V sends e, P sends $k + ex$

This will allow V to check using the curve points, as we've already discussed, if he has the public key for x - using $(R + eP)$. But since he can't get k, P is protected from V learning x via simple arithmetic.

So now we're going to see how this pattern plays out in this more complex case.

4.2 The commitment step for the inner product proof

The Prover P will need to send *two* commitments, analogous to the R value in the above description – one for each of $\mathbf{x}$ and $\mathbf{y}$. These will be called $\mathbf{d}_x, \mathbf{d}_y$ respectively. But there's another difference – in our stated problem, we have Pedersen commitments to the vectors, rather than the vectors themselves (this is analogous to how, in the Schnorr protocol, you have a public key P, not the secret x, so you have to send the nonce-*point* R, not the raw nonce itself, k), so instead of sending $\mathbf{d}_x, \mathbf{d}_y$, the Prover will instead send Pedersen commitments to them:

$$A_d = r_d H + \mathbf{d}_x \mathbf{G}$$
$$B_d = s_d H + \mathbf{d}_y \mathbf{G}$$

r_d, s_d will be random values as usual for Pedersen commitments.

To leverage the analogy further, just as the final Schnorr response is $ex + k$, so here our final response(s) are of the form $e\mathbf{x} + \mathbf{d}$, more specifically, one for each: $e\mathbf{x} + \mathbf{d}_x, e\mathbf{y} + \mathbf{d}_y$.

However, that's not enough; we're trying to prove an inner product, too.

What we'll have to do also in the challenge step is to send a *commitment* to the expected inner product of this *blinded* form of our vectors. The blinded form has already been mentioned as $e\mathbf{x} + \mathbf{d}_x, e\mathbf{y} + \mathbf{d}_y$, but we don't yet know the challenge e, so we have to factor that out somehow.

Now, e is a linear factor in each of these terms, so dot-product-ing them$((e\mathbf{x}+\mathbf{d}_x) \cdot (e\mathbf{y}+\mathbf{d}_y))$ will result in a quadratic in e, so there will be three coefficients, and we'll therefore need to provide commitments in advance for each of these three coefficients. However, we already have the coefficient of e^2; that's C_z, which was given in advance. So we'll therefore need to provide *two* additional commitments:

$$C_1 = t_1 H + (\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x) G$$
$$C_0 = t_0 H + (\mathbf{d}_x \cdot \mathbf{d}_y) G$$

(Note the use of G not $\mathbf{G}$ because dot products are scalars, not vectors).

So to do all this in the commitment step, the Prover had to come up with 4 random scalars r_d, s_d, t_1, t_0 and two random vectors $\mathbf{d}_x, \mathbf{d}_y$ and then send 4 Pedersen commitments using this data: A_d, B_d, C_1, C_0.

4.3 The challenge step

Nothing to discuss here – the Verifier simply sends a single scalar value e.

4.4 The response step

The above detailed discussion will hopefully make the following set of data, sent by the Prover, less bewildering:

$$\begin{aligned}
\mathbf{f}_x &= e\mathbf{x} + \mathbf{d}_x \\
\mathbf{f}_y &= e\mathbf{y} + \mathbf{d}_y \\
r_x &= er + r_d \\
s_y &= es + s_d \\
t_z &= e^2 t + et_1 + t_0
\end{aligned}$$

First, note that here we are sending the blinded forms $\mathbf{f}_x, \mathbf{f}_y$ of the two vectors, not the Pedersen commitments – the idea is that the Verifier will verify precisely by reconstructing the commitments and checking they match C_x, C_y. Those two checks are:

$$\begin{aligned}
eC_x + A_d &=? \; r_x H + \mathbf{f}_x \mathbf{G} \\
eC_y + B_d &=? \; s_y H + \mathbf{f}_y \mathbf{G}
\end{aligned}$$

The r_x, s_y were chosen to make the random values equate in the relevant Pedersen commitments. It's not hard to see that these equations will verify for honest behaviour (remember, this is called "completeness"):

$$\begin{aligned}
&eC_x + A_d \\
&= e(rH + \mathbf{x}\mathbf{G}) + r_d H + \mathbf{d}_x \mathbf{G} \\
&= (er + r_d)H + (e\mathbf{x} + \mathbf{d}_x)\,\mathbf{G} \\
&= r_x H + \mathbf{f}_x \mathbf{G}
\end{aligned}$$

, and the same for the $\mathbf{y}$ equations.

The last line, for t_z, is needed for the third check - the one that ensures that the inner product is correct, and it ensures the random values in the Pedersen commitment equation will be correct. It was for this, remember, that we sent the two commitments C_1, C_0. The check is really that:

$$\mathbf{f}_x \cdot \mathbf{f}_y = e^2 z + e\,(\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x) + \mathbf{d}_x \cdot \mathbf{d}_y \quad \because z = \mathbf{x} \cdot \mathbf{y}$$

But this equation is reconstructed "under" Pedersen commitments; using Comm as shorthand for that, we want the verifer to be able to reconstruct a check that:

$$\mathrm{Comm}\,(\mathbf{f}_x \cdot \mathbf{f}_y) = e^2 C_z + e\,(\mathrm{Comm}\,(\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x)) + \mathrm{Comm}\,(\mathbf{d}_x \cdot \mathbf{d}_y)$$

, but we prepared C_1, C_0 and t_z to fulfil exactly this role, so we have

$$t_z H + (\mathbf{f}_x \cdot \mathbf{f}_y)\,G =? \; e^2 C_z + eC_1 + C_0$$

and the reader can now easily check for himself, by simple substitution, that this will be passed in the honest case.

4.5 Knowledge soundness

What we need to prove here is not only that the Prover knows $\mathbf{x}, \mathbf{y}$, but also that $z = \mathbf{x} \cdot \mathbf{y}$. So our Extractor is tasked with extracting those vectors and must be able to verify that the dot product equation holds.

First let's note the transcript: it's $((A_d, B_d, C_1, C_0), e, (\mathbf{f}_x, \mathbf{f}_y, r_x, s_y, t_z))$. As a reminder: the 'commitment' step of the Sigma protocol involves sending those 4 curve points (each of which are themselves Pedersen commitments), then the Verifier sends back a single scalar challenge, then the 'response' step involves the Prover sending two blinded vectors (f), along with random scalars to make the Pedersen commitments verify correctly, and finally the t_z to allow the completed verification of the inner product.

What can we do if the Extractor gets the Prover to output a second accepting transcript with the same commitment, say $\big((A_d, B_d, C_1, C_0), e', (\mathbf{f}'_x, \mathbf{f}'_y, r'_x, s'_y, t'_z)\big)$?

Let's isolate our consideration to the $\mathbf{x}$-part only: we have two validating checks, one from each transcript:

$$eC_x + A_d = r_x H + \mathbf{f}_x \mathbf{G}$$
$$e'C_x + A_d = r'_x H + \mathbf{f}'_x \mathbf{G}$$

Subtracting the second equation from the first and multiplying both sides by $\eta = (e - e')^{-1}$ gives:

$$C_x = \eta(r_x - r'_x)H + \eta(\mathbf{f}_x - \mathbf{f}'_x)\mathbf{G}$$

and we note by the same logic as for the set-of-vectors case, that this, by the binding property of the Pedersen commitment, proves that

$$\mathbf{x} = \eta(\mathbf{f}_x - \mathbf{f}'_x)$$

We can also now extract the opening of the commitment A_d: it's just $\mathbf{d}_x = \mathbf{f}_x - e\mathbf{x}$, for example.

... and the same procedure can of course extract $\mathbf{y}, \mathbf{d}_y$.

However, to go to the final step, we also need to get the opening of the commitment C_z to z. This is a little more involved but follows the same concept: take the two transcripts, cancel then rearrange the equation so that commitment is the subject, and observe the content of the coefficient of G, using the binding property observe that that must be the opening (and making use of the ability

to cross multiply by an inverse scalar):

$$
\begin{aligned}
&t_z H + (\mathbf{f}_x \cdot \mathbf{f}_y)\, G = e^2 C_z + eC_1 + C_0 \quad (1)\\
&t'_z H + \left(\mathbf{f}'_x \cdot \mathbf{f}'_y\right) G = e'^2 C_z + e' C_1 + C_0 \quad (2)\\
&(1) - (2)\ ;\ \text{and consider G coefficient :}\\
&\because C_z = tH + zG\ ;\ C_1 = t_1 H + (\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x)\, G \therefore\\
&\left(\mathbf{f}_x \cdot \mathbf{f}_y - \mathbf{f}'_x \cdot \mathbf{f}'_y\right) G = \left(e^2 - e'^2\right) zG + (e - e')\,(\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x)\, G\\
&\implies\\
&\left(e^2 - e'^2\right) zG = \left(\left(\mathbf{f}_x \cdot \mathbf{f}_y - \mathbf{f}'_x \cdot \mathbf{f}'_y\right) - (e - e')\,(\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x)\right) G\\
&\text{Let } \gamma = \left(e^2 - e'^2\right)^{-1}\ :\\
&z = \gamma \left(\left(\mathbf{f}_x \cdot \mathbf{f}_y - \mathbf{f}'_x \cdot \mathbf{f}'_y\right) - (e - e')\,(\mathbf{x} \cdot \mathbf{d}_y + \mathbf{y} \cdot \mathbf{d}_x)\right)
\end{aligned}
$$

Proving that $z = \mathbf{x} \cdot \mathbf{y}$ is a little different. Note that, we *expect* of course that this equation is true, but the above argument doesn't completely *prove* it; it is only an algorithm to extract the specific set of values for $z, \mathbf{x}, \mathbf{y}$ that were committed to. Proof basically amounts to observing that in the polynomial equation in the challenge e, for any one specific transcript, we expect the coefficients to be equal:

$$(e\mathbf{x} + \mathbf{d}_x) \cdot (e\mathbf{y} + \mathbf{d}_y) = ze^2 + z_1 e + z_0$$

(here we boil down the *openings* of C_1, C_0 as z_1, z_0 for readability). The paper appeals to the Schwartz-Zippel lemma (see 3.3), which states (in simple terms) that the probability of two *different* coefficient sets satisfying the equality must be less than d/p where d is the degree of the polynomial; since here $d = 2$ and p is a very large prime, this is basically negligible. We thus conclude that $z = \mathbf{x} \cdot \mathbf{y}$ (this is the first coefficient, of e^2, of course) must hold, with overwhelming probability.

4.6 Zero-knowledgeness

The argument here will be the same basic idea as for the set-of-vectors case (and similarly sketchy, I won't include detail).

As in previous versions of this argument, we set ourself the task of generating a fake transcript with the same distribution. And as before (unsurprisingly, as it was explained in some painstaking detail in the previous section that this retains a lot of the "spirit" of the Schnorr-style sigma protocol), the way this is achieved is generating the elements of the transcript out of order.

C_z, C_x, C_y will be set in advance; will be treated as an input. The Simulator will then pick randomly the response section of the transcript: $(\mathbf{f}_x, \mathbf{f}_y, r_x, s_y, t_z)$ (if you've forgotten the structure of the transcript, read again the first part of the previous section). It'll also choose the commitment C_1 as a commitment to zero (note that because Pedersen commitments have randomness, that's still blinded and random). This is enough for it to reconstruct the first part of the

transcript, as follows:

$$A_d = (r_x H + \mathbf{f}_x \mathbf{G}) - eC_x$$
$$B_d = (s_y H + \mathbf{f}_y \mathbf{G}) - eC_y$$

By substitution you can see these satisfy the definitions given originally for A_d, B_d. Similarly you can reconstruct what C_0 must be:

$$C_0 = t_z H + (\mathbf{f}_x \cdot \mathbf{f}_y)\, G - eC_1 - e^2 C_z$$

Verifier will check:

$$e^2 C_z + eC_1 + C_0 =? \; t_z H + \mathbf{f}_x \cdot \mathbf{f}_y G$$
$$\text{Substituting defn of } \; C_0 : \text{LHS} =$$
$$e^2 C_z + eC_1 + t_z H + \mathbf{f}_x \cdot \mathbf{f}_y G - eC_1 - e^2 C_z$$
$$= \text{RHS}$$

Hence the faked transcript

$$((A_d = (r_x H + \mathbf{f}_x \mathbf{G}) - eC_x,$$
$$B_d = (s_y H + \mathbf{f}_y \mathbf{G}) - eC_y,$$
$$C_1 = t_1 H + 0G,$$
$$C_0 = t_z H + (\mathbf{f}_x \cdot \mathbf{f}_y)\, G - eC_1 - e^2 C_z),$$
$$e, (\mathbf{f}_x, \mathbf{f}_y, r_x, s_y, t_z))$$

will verify, by this construction.

It's not hard to be convinced (although I won't try formally) that the real and faked transcripts will both have random distributions – the important point being that these Pedersen commitments all have randomness included for the hiding property.

5 A more compact inner product proof

The paper by Bootle et al. [2] has a more sophisticated approach to this problem – fiendishly clever in fact – involving recursion; and indeed, this method is very similar to the idea used in Bulletproofs; it is its direct precursor. It's also worth mentioning that it's inspired by this Groth paper [21], although that version didn't include the critical recursion idea.

The goal here is principally to reduce the amount of data communicated between the two parties (and when it's switched to a non-interactive form, using the Fiat Shamir heuristic, that means a more compact proof).

Note that in this form, there is no attempt to make the argument-of-knowledge be zero-knowledge (so in itself, it only has the compactness advantage).

5.1 Condensing a single vector

Before getting into the inner product aspect, we can start with a more basic goal – reduce how much data we have to send in order to just commit and then prove knowledge of a vector. Let's start by considering a vector of dimension 10, as a concrete example. We'll use a Pedersen commitment to the vector, but with no randomness; we'll write it here as a starting point for the discussion:

$$A = a_1G_1 + a_2G_2 + \ldots + a_{10}G_{10}$$

Straightforwardly revealing $[a_1, a_2, \ldots, a_{10}]$ would suffice to prove knowledge of course; but this requires 10 scalar values (which may be 32 bytes each in typical elliptic curve scenarios) as well as the original commitment. Is there a way to condense down the "revelation" part?

Revealing the sum $a_1 + a_2 + \ldots + a_{10}$ is, trivially, a dumb idea since there are many different combinations of 10 numbers that could give that same sum.

The clever trick employed is this – to transform the commitment A to a different commitment A', which commits to a vector with a smaller number of elements, but also *contains* the original commitment A! We start by chopping the original vector into equal sized pieces:

$$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}] \rightarrow [[a_1, a_2], [a_3, a_4], [a_5, a_6], [a_7, a_8], [a_9, a_{10}]]$$

and use the same chopping/chunking operation on the G_is also. We're going to use the following notation for the RHS of the above: $[\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4, \mathbf{a}_5]$, i.e. 5 vectors of dimension 2. The G-part will be written likewise: $[\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3, \mathbf{G}_4, \mathbf{G}_5]$. (Note: the particular choice here just depends on the factorization; we had 10 here, so we chose 5x2. If we started with 21 we'd have to choose 7 and 3, etc.). We can visualize this new arrangement in matrix form:

$$\begin{pmatrix} \mathbf{a}_1\mathbf{G}_1 & \mathbf{a}_2\mathbf{G}_1 & \ldots & \ldots & \mathbf{a}_5\mathbf{G}_1 \\ \mathbf{a}_1\mathbf{G}_2 & \mathbf{a}_2\mathbf{G}_2 & \ldots & \ldots & \ldots \\ \ldots & \ldots & \mathbf{a}_3\mathbf{G}_3 & \ldots & \ldots \\ \ldots & \ldots & \ldots & \mathbf{a}_4\mathbf{G}_4 & \ldots \\ \mathbf{a}_1\mathbf{G}_5 & \ldots & \ldots & \ldots & \mathbf{a}_5\mathbf{G}_5 \end{pmatrix}$$

We note the main diagonal particularly, since its sum is in fact A, because:

$$\mathbf{a}_i\mathbf{G}_i = a_{2i-1}G_{2i-1} + a_{2i}G_{2i} \quad \forall i \in 1..5$$

The Prover is going to send commitments to the *diagonals* of this matrix; we'll see in a minute why this is particularly useful; note that the total number of such commitments is $2 \times 5 - 1$, including the main diagonal which is already known as A. The formula for the diagonals is:

$$A_k = \sum_{\max(1,1-k)}^{\min(5,5-k)} \mathbf{a}_{i+k}\mathbf{G}_i \quad \text{for} \quad k = -4, -3..0..3, 4$$

It's pretty fiddly but if you work through carefully, you'll see that this will provide 9 $(= 2 \times 5 - 1)$ commitments, one for each of the diagonals of the above matrix. For example, set $k = -4$, we get lower limit = upper limit of sum = 5,

so we have $\mathbf{a}_1\mathbf{G}_5$, which is precisely the lower left entry in the matrix, i.e. the first of the set of diagonals, travelling up and to the right.

At this point we receive a challenge, x, from the Verifier.

The next step is where the actual "condensing" happens. Create new vectors **of dimension 2** that are defined like this:

$$\mathbf{a}' = \sum_{i=1}^{5} x^i \mathbf{a}_i \quad \mathbf{G}' = \sum_{i=1}^{5} x^{-i} \mathbf{G}_i$$

This is tricky notation wise (remember our convention that we bold "vectors" of curve points), so let's be explicit:

$$\begin{aligned}\mathbf{G}' &= [x^{-1}G_1, x^{-1}G_2] + [x^{-2}G_3, x^{-2}G_4] + [x^{-3}G_5, x^{-3}G_6] + [x^{-4}G_7, x^{-4}G_8] + [x^{-5}G_9, x^{-5}G_{10}] \\ &= [\left(x^{-1}G_1 + x^{-2}G_3 + x^{-3}G_5 + x^{-4}G_7 + x^{-5}G_9\right), \left(x^{-1}G_2 + x^{-2}G_4 + x^{-3}G_6 + x^{-4}G_8 + x^{-5}G_{10}\right)]\end{aligned}$$

... with the same basic pattern for $\mathbf{a}'$(except the powers of x are positive). Now we recreate the commitment for this new "coordinate system":

$$\begin{aligned}A' &= \mathbf{a}'\mathbf{G}' \\ &= (xa_1 + x^2a_3 + x^3a_5 + x^4a_7 + x^5a_9)G'_1 + (xa_2 + x^2a_4 + x^3a_6 + x^4a_8 + x^5a_{10})G'_2\end{aligned}$$

where G'_1, G'_2 are the 2 components of $\mathbf{G}'$ defined above. Now this multiplication has, not accidentally, a specifically useful cancellation: the powers of x cancel where the indices of a, G match. For maximum clarity, we visualize this again as a matrix:

$$\begin{pmatrix} a_1G_1 + a_2G_2 & x(a_3G_1 + a_4G_2) & x^2(a_5G_1 + a_6G_2) & x^3(a_7G_1 + a_8G_2) & x^4(a_9G_1 + a_{10}G_2) \\ x^{-1}(a_1G_3 + a_2G_4) & a_3G_3 + a_4G_4 & x(a_5G_3 + a_6G_4) & x^2(a_7G_3 + a_8G_4) & x^3(a_9G_3 + a_{10}G_4 \\ \dots & \dots & a_5G_5 + a_6G_6 & \dots & \dots \\ \dots & \dots & \dots & a_7G_7 + a_8G_8 & \dots \\ x^{-4}(a_1G_9 + a_2G_{10}) & \dots & \dots & \dots & a_9G_9 + a_{10}G_{10} \end{pmatrix}$$

Note, these are not literally matrices we're looking at, but just a visualization of the multiplication of the two terms. We see that, as before the inclusion of x, the main diagonal is exactly the entirety of $A = a_1G_1 + a_2G_2 + \ldots + a_{10}G_{10}$.

Now we see, however, the importance specifically of the diagonals, as opposed to other slicings-up of the matrix: the diagonals are the sets of terms which are multiplied by the same power of x. To validate the correspondence between A and A', a Verifier will need to check that the entire set ("matrix") of terms; to do this he checks whether:

$$A' = \sum_{\max(1,1-k)}^{\min(5,5-k)} x^k A_k = \sum_{\max(1,1-k)}^{\min(5,5-k)} x^k \mathbf{a}_{i+k}\mathbf{G}_i \quad \text{for} \quad k = -4, -3..0..3, 4$$

He only checks the left equation of course; the full decomposition is known only to the Prover.

Imagining this as an interaction; it's a little confusing since it *only* has any point in cases where we "recurse":

The Prover asserts that A is a commitment to $\mathbf{a}$, and then sends the full list $A_{-4}, A_{-3}, \ldots, A_0 = A, \ldots, A_4$. Then the verifier replies with a challenge x. Unlike a typical Sigma protocol, the Prover does not then send back a proof. What happens instead is, **both sides now construct a reduced form of the same problem**. (Aside: Jonathan Bootle illustrated this idea helpfully

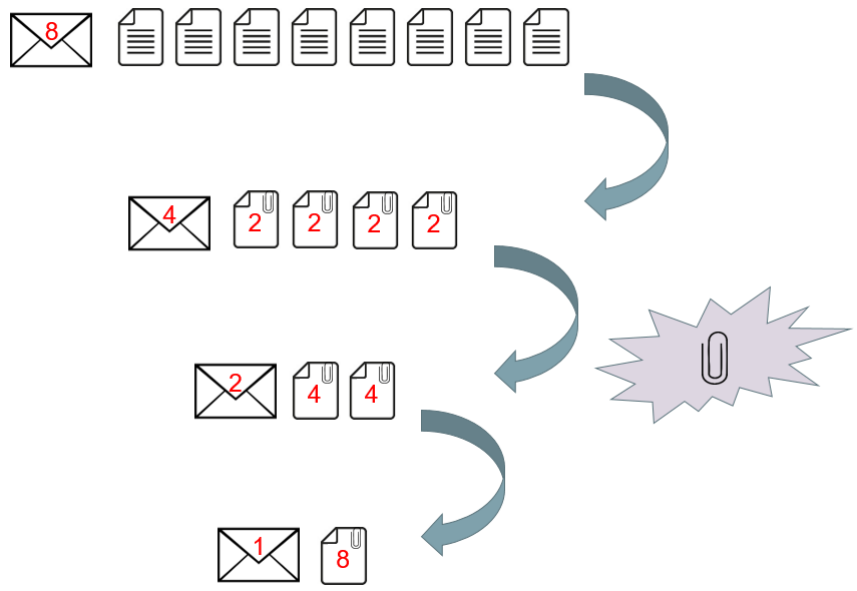

Figure 3: *Compressing commitments - attribution of this image below*

in this [4]blog post – see Figure 3, taken from that post – this gives you the mental model of what's going on; we're repeatedly shrinking the size of the proof we'll have to finally create before the final step (the envelopes represent commitments)):

Both Prover and Verifier can construct $A', \mathbf{G}'$ (while only the Prover of course constructs $\mathbf{a}'$), and they now are back in the starting position with smaller vectors – in our concrete example, they started with a vector of dimension 10, it's now a vector of dimension 2. If the process is to end there, the Prover will simply reveal this vector (here it would be the components $(xa_1 + x^2a_3 + x^3a_5 + x^4a_7 + x^5a_9), (xa_2 + x^2a_4 + x^3a_6 + x^4a_8 + x^5a_{10})$, as two numbers).

But if we started with, say, a vector of dimension 600, we could have followed the above procedure as 60x10, and at this point the basis vectors $\mathbf{G}'$ may have dimension 60, and we could repeat, chunking 60 = 6 x 10 again. We would repeat the above algorithm, except replacing $A \to A'$, $\mathbf{G} \to \mathbf{G}'$.

Summarizing: in the case with multiple reductions, the pattern of interaction would be : P sends diagonal commitments, V sends x, (both sides calculate next step), P sends new diagonal commitments, V sends x', ... , last step: P sends full commitment openings to a small vector.

So has this admittedly complicated algorithm helped us? Let's count: for the dimension-10 case, first we had to send an additional 9 - 1 commitments (subtract off the main diagonal; that's A, which we transferred at the start),

then we revealed 2 scalars at the end. So that's actually ~ the same as the 10 scalars we'd have to reveal if we did it without shenanigans.

But for the case $600 = 10 \times 10 \times 6$ - we first "chunk" in 10s, then again in 10s, leaving only 6 components for the final step. That requires revealing $2 \times 10 - 1 = 18$ commitments at each of the two reducing steps, along with 6 scalars in the final step (and again subtract 1 for the starting A). That'd be only 43 items instead of 600.

So far so good – it clearly saves space, but we've left a big hole in the argument; does this actually have *knowledge soundness*?

5.2 Knowledge soundness of the argument

We can isolate this to the question: does the provision of the diagonal commitments A' (strictly, the provision of the A values, from which the A' are derived using x), in any step, prove knowledge of the uncondensed vector $\mathbf{a}$? Because (a) trivially the final step has soundness; it reveals the vector in its entirety and (b) all of the reducing steps can be treated the same (with different dimension of course; here we're sticking to the 10→2 reduction for concreteness).

As in the first section of the document, we have a situation where the Extractor collecting a "complete set" of equations for multiple challenges (in this case, $2 \times 5 = 1$ values of x) will allow us to define a complete solution to a set of linear equations, although here it's meaningfully more complicated!

For each such challenge x, an interaction will provide an equation like:

$$\sum_{k=-4}^{4} x^k A_k = \mathbf{a}'\mathbf{G}' = \mathbf{a}' \sum_{i=1}^{5} x^{-i} \mathbf{G}_i$$

Remember that the publically shared information here is the A_ks, the x and the $\mathbf{G}_i$s. The Extractor-Verifier therefore knows $\mathbf{G}'$; if he can extract the set of coefficients of $\mathbf{G}$ he can thus open the commitment to $\mathbf{a}'$ which the LHS of the above equation represents (remember, such a sum of commitments is actually just one curve point).

If he can reconstruct that opening in the form: $\mathbf{a}' = \sum_{i=1}^{5} \mathbf{a}_i x^i$, for some set $\mathbf{a}_i$, then we have a valid opening of the commitment and the Extractor's job is done (he has extracted the original vector committed to).

As in the first section, on proof of knowledge of a set of vectors, because we chose x-powers as multiplying factors, we have a Vandermonde matrix (actually a *shifted* Vandermonde matrix, because we used negative as well as positive powers), and that ensures we can invert it (to put it another way, as the paper does: it's guaranteed that, as long as the challenges x are all distinct, you'll be able to find a particular linear combination of the equations that isolates the formula for one particular A_k). Choosing one particular A_k, say A_{-4}, then by such isolation, you can get a formula $A_{-4} = \sum_{i=1}^{5} \mathbf{a}_{-4,i} \mathbf{G}_i$, where $\mathbf{a}_{-4,i}$is some linear combination of the original vectors $\mathbf{a}$ - note that this procedure has *explicitly calculated* these $\mathbf{a}_{-4,i}$s. We'll do this for each of the 9 A_ks; consider what

happens if you apply this new formula to the original equation:

$$A_k = \sum_{i=1}^{5} \mathbf{a}_{k,i}\mathbf{G}_i \quad \forall k \in -4 \dots 4$$
$$\therefore \sum_{k=-4}^{4} x^k \left(\sum_{i=1}^{5} \mathbf{a}_{k,i}\mathbf{G}_i \right) = \mathbf{a}'\mathbf{G}' = \mathbf{a}' \sum_{i=1}^{5} x^{-i}\mathbf{G}_i$$
$$\therefore x^{-i}\mathbf{a}' = \sum_{k=-4}^{4} \mathbf{a}_{k,i}x^k \quad \forall i \in 1 \dots 5$$

where the last deduction specifically arises from the fact that the coefficients of the $\mathbf{G}_i$s must be equal. Now note that this has given us 5 different equations for $\mathbf{a}'$, in terms of a 9 by 5 matrix of vector coefficients $\mathbf{a}_{k,i}$. By comparing coefficients of powers of x, you can see that in fact $\mathbf{a}_{k,i} = \mathbf{a}_{k+i}$ if $k + i \in 1..5$ and is zero otherwise – the proof is left as an exercise for the reader (in other words, I haven't figured it out yet!), but it's easy to check by hand. So finally the relation boils down to:

$$\mathbf{a}' = \sum_{i=1}^{5} \mathbf{a}_i x^i$$

which proves that the $\mathbf{a}_i$s we calculated (remember we went from A_k to $\mathbf{a}_{k,i}$ to $\mathbf{a}_{k+i}$ to the $\mathbf{a}_i$ in the above calculation, in other words, we explicitly calculated them from a matrix inversion) are actually exactly those committed to.

5.3 Extending to an inner product

It might seem like we've barely started, since we only talked about one vector $\mathbf{a}$ so far. However, the power of the above algorithm extends almost directly to the more general goal of proving knowledge of: $\mathbf{a}, \mathbf{b}, z$ such that $z = \mathbf{a} \cdot \mathbf{b}$.

First, one can repeat the exact algorithm above for the knowledge of $\mathbf{b}$; although to make the inner product work, we modify it in two trivial ways: we replace $\mathbf{G}$ with $\mathbf{H}$ (i.e. different base curve points in the commitments), and we replace the challenge x with its inverse x^{-1}.

So this would require commitments to the diagonals as before, a total of $2m-1$ commitments B_k (we'll now switch from a concrete structure $10 = 5 \times 2$, to a generic $n = m_1 \times m_2 \dots$, i.e. a factorization of some dimension n), and we will have:

$$B' = \sum_{k=1-m}^{m-1} B_k x^{-k}$$

It remains to consider the $z = \mathbf{a} \cdot \mathbf{b}$ part. Remember that we have "chunked" both these vectors, like so: $\mathbf{a} = [\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \mathbf{a}_4, \mathbf{a}_5]$ and $\mathbf{b} = [\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5]$; we take the same approach of committing to diagonals, this time of $\mathbf{a} \cdot \mathbf{b}$ in the matrix:

$$\begin{pmatrix} \mathbf{a}_1\mathbf{b}_1 & \mathbf{a}_2\mathbf{b}_1 & \dots & \dots & \mathbf{a}_5\mathbf{b}_1 \\ \mathbf{a}_1\mathbf{b}_2 & \mathbf{a}_2\mathbf{b}_2 & \dots & \dots & \dots \\ \dots & \dots & \mathbf{a}_3\mathbf{b}_3 & \dots & \dots \\ \dots & \dots & \dots & \mathbf{a}_4\mathbf{b}_4 & \dots \\ \mathbf{a}_1\mathbf{b}_5 & \dots & \dots & \dots & \mathbf{a}_5\mathbf{b}_5 \end{pmatrix}$$

We have already defined $\mathbf{a}' = \sum_{i=1}^{m} \mathbf{a}_i x^i$, we now define $\mathbf{b}' = \sum_{i=1}^{m} \mathbf{b}_i x^{-i}$ (note that the role of x has to be reversed to create the same cancellation as before). As before, we define commitments to diagonals, and construct:

$$z' = \sum_{k=1-m}^{m-1} z_k x^k, \quad z_k = \sum_{i=\max(1,1-k)}^{\min(m,m-k)} \mathbf{a}_i \cdot \mathbf{b}_{i+k}$$

and it's easy to see that we have the same pattern as the previous section: $z' = \mathbf{a}' \cdot \mathbf{b}'$.

So finally we see that the reduction step previously mentioned will work for proof of the inner product as well as the vectors themselves.

To sum up: the Prover sends initially (A, B, z) (assuming the vectors $\mathbf{G}, \mathbf{H}$ are set in advance), along with the factorization of the dimension n into $(m_1, m_2, \ldots)$, then for each reduction step sends $(A_k, B_k, z_k \; \forall k \in \max(1, 1-k) \ldots \min(m, m-k))$, while of course the Verifier sends a challenge x for each step. And as noted before the final step simply involves revealing the final two vectors $\mathbf{a}', \mathbf{b}'$, whose dot product will be z'.

Note: in the interest of brevity, I've for now omitted including the logic for the soundness of the inner product part of the proof.

5.4 Scaling

The great achievement of this construction is that its communication cost between the Prover and Verifier is "basically" logarithmic in the dimension n of the vectors $\mathbf{a}, \mathbf{b}$.

For any arbitrary vectors dimension n, whose factorization is $\prod_i^{\alpha} m_i$, the cost is about $6 \prod_{i=2}^{\alpha} (m_i - 1)$, so, linear in the sum of the factors. A power of 2 (as was illustrated in the diagram from Bootle's blog post), will be a somewhat optimal case (the more factors the better) –we will have one reduction step for each power of 2, i.e. k steps for dimension 2^k, and so the cost will be $\simeq 6 \log_2 n$.

6 Bulletproofs

The ideas laid out in the Bulletproofs[15] paper by Bünz et al. are basically (a) an even more compact version of the inner product argument of knowledge, (b) how to construct a compact rangeproof using such an argument of knowledge (c) how to generalize this idea to general arithmetic circuits. We'll focus on (a) and (b), and only briefly mention (c).

6.1 An even more compact inner product proof

The general flavour of the arguments laid out in Section 3 of the Bulletproofs paper is, how can we most efficiently make use of the essential/basic element of the very powerful reduction/recursion argument laid out in detail above?

As before, start by considering a single vector ... but wait, in Section 3, Bünz goes back (helpfully!) even a step further and considers just committing to 2 scalars a, b. Let's say we commit to them with a commitment $C = aG_1 + bG_2$ (as in the previous section we are omitting blinding). If you wanted to fold this commitment together so as to reveal only *one* scalar in the commitment opening,

you'd need to somehow combine a and b together. As we've already observed at least once, "combining" values under commitment in a naive way loses the binding property – a commitment to $a+b$ is useless as it might just as easily be a commitment to $(a+\alpha),(b-\alpha)$ as to (a,b). A commitment to something like $(ax+b)$, with x being the challenge as per usual, seems like a step up – but how is the verifier going to verify the commitment? $C(ax+b) = xC(a)+C(b)$ by linearity, but that is not a function of the original commitment C. We need a function $f(a,b,x)$ from these three values to a single scalar a', which, when combined with a function $g(G_1,G_2,x)$ from the basepoints and x to a new basepoint G', such that we can construct a commitment verifier-side. This construction is:

$$
\begin{aligned}
&a' = ax + bx^{-1}, \quad G' = x^{-1}G_1 + xG_2 \\
&\therefore C' = a'G' = \left(ax + bx^{-1}\right)\left(x^{-1}G_1 + xG_2\right) \\
&= aG_1 + bG_2 + x^2aG_2 + x^{-2}bG_1 = C + x^2L + x^{-2}R
\end{aligned}
$$

where C was the original commitment.

This little example illustrates a central idea, but in itself is of course useless – you would have to open the commitment using a', L, R instead of just a, b which is longer not shorter. So the point here is what happens when we scale it up to vectors, in particular, long ones.

So the next step is to look at a vector like $a_1, a_2, \ldots, a_n$ instead of a, b. The commitment would be as according to previously used notation; here $\mathbf{aG}$. What you do is pair off the entries in the first half and the second half of the vector to re-create the above effect, but crucially, **you still only need one each of L and R**:

$$
\begin{aligned}
&(a_1, a_2, \ldots, a_{n/2}), (a_{n/2+1}, a_{n/2+2}, \ldots, a_n) \quad \text{“cut”} \\
&\rightarrow \\
&([a_1, a_{n/2+1}], [a_2, a_{n/2+2}], \ldots, [a_{n/2}, a_n]) \quad \text{“fold”} \\
&\leftarrow \text{Verifier sends } x \\
&(xa_1 + x^{-1}a_{n/2+1}, xa_2 + x^{-1}a_{n/2+2}, \ldots, xa_{n/2} + x^{-1}a_n)
\end{aligned}
$$

where the first two steps "cut" and "fold" are just to illustrate the concept (they're not mathematical operations, just rearrangements of the vector). Now note that if we perform the **same** cut-and-fold operation on the basepoints $\mathbf{G}$, with the exception that we apply the reciprocal of the challenge, then we'll get:

$$
\mathbf{G}' = (x^{-1}G_1 + xG_{n/2+1}, x^{-1}G_2 + xG_{n/2+2}, \ldots, x^{-1}G_{n/2} + xG_n)
$$

so that

$$
\mathbf{a}'\mathbf{G}' = (xa_1 + x^{-1}a_{n/2+1})(x^{-1}G_1 + xG_{n/2+1}), (xa_2 + x^{-1}a_{n/2+2})(x^{-1}G_2 + xG_{n/2+2}), \ldots, \\
(xa_{n/2} + x^{-1}a_n)(x^{-1}G_{n/2} + xG_n)
$$

which reduces to

$$\begin{aligned}\mathbf{a}'\mathbf{G}' &= \mathbf{a}\mathbf{G} + x^2\left(a_1 G_{n/2+1} + a_2 G_{n/2+2} + \ldots + a_{n/2} G_n\right) + \\ &\quad x^{-2}\left(a_{n/2+1} G_1 + a_{n/2+2} G_2 + \ldots + a_n G_{n/2}\right) \\ &= \mathbf{a}\mathbf{G} + x^2 L + x^{-2} R\end{aligned}$$

with L, R defined as the parenthesized terms.

So the interaction (a similar pattern to what we saw with Bootle with the matrix diagonals) is: Prover constructs and sends L, R, Verifier sends x, then both sides can construct $\mathbf{G}'$ and reconstruct the final $C' = \mathbf{a}'\mathbf{G}'$ as above.

Note that we already have the key scalability win: we've reduced the communication from n terms to $n/2 + 2$ (the extra two for L, R). And if we apply the protocol repeatedly, we get to $2 + \log_2 n \times 2$ (because we need a new for each halving step, and a final reveal of two scalars).

6.1.1 Two vectors in parallel

It's trivial to do this with commitments to two vectors, $C = \mathbf{a}\mathbf{G} + \mathbf{b}\mathbf{H}$. Assume they both have dimension n, a power of 2. The construction composes directly; you can simply add the L, Rs for each one; because they are using different base curve points, and the pattern holds in the same way:

$$\begin{aligned}L = L_a + L_b = &\left(a_1 G_{n/2+1} + a_2 G_{n/2+2} + \ldots + a_{n/2} G_n\right) + \\ &\left(b_1 H_{n/2+1} + b_2 H_{n/2+2} + \ldots + b_{n/2} H_n\right)\end{aligned}$$

and the same for R.

In other words, the Prover starts with commitment $\mathbf{a}\mathbf{G} + \mathbf{b}\mathbf{H}$, and at each halving iteration sends across the composed values of L, R; everything is as before. To be explicit, the "reduced"/ "halved" form of the commitment is:

$$C' = \mathbf{a}'\mathbf{G}' + \mathbf{b}'\mathbf{H}' = C + x^2(L_a + L_b) + x^{-2}(R_a + R_b)$$

6.1.2 Re-introducing the inner product

Bünz's paper separates this into two parts. First, he modifies the above construction to include an explicit value of the inner product of the two vectors, then he overlays a simple additional step to enforce that overlay explcitly(see "Protocol 1" and "Protocol 2"; Protocol 2 is the actual proof while Protocol 1 enforces the value of the inner product and calls Protocol 2 as a subroutine).

If the Prover asserts that the commitment $C = zG + \mathbf{a}\mathbf{G} + \mathbf{b}\mathbf{H}$ (minor technical note: the G basepoint for z is not one of the elements of the vector $\mathbf{G}$; it can be another curve point such as the curve's known generator), is in fact a commitment to the inner product as well as the two vectors, then it is not too difficult to extend the above construction to include the ability for the Verifier to verify that $z = \mathbf{a} \cdot \mathbf{b}$. We just need to ensure that, at each halving step, $z' = \mathbf{a}' \cdot \mathbf{b}'$ (remember – the whole idea here is to reduce one version of the problem to a smaller one, meaning we have to preserve all the properties at each step). Let's first note what we get from simply reconstructing the new

commitment from the old one, without any modifications to the algorithm from the previous section:

$$C' = C + x^2 L + x^{-2} R = zG + \mathbf{aG} + \mathbf{bH} + x^2 L + x^{-2} R$$

The problem now is that z is no longer a dot product of the reduced sized vectors $\mathbf{a}', \mathbf{b}'$; it's just the dot product of the original vectors, which is different. That new dot product is:

$$\begin{aligned}
&(xa_1 + x^{-1}a_{n/2+1}, xa_2 + x^{-1}a_{n/2+2}, \ldots, xa_{n/2} + x^{-1}a_n)\cdot \\
&(x^{-1}b_1 + xb_{n/2+1}, x^{-1}b_2 + xb_{n/2+2}, \ldots, x^{-1}b_{n/2} + xb_n) \\
&= a_1b_1 + a_2b_2 + \ldots + a_nb_n + \\
&x^2(a_1b_{n/2+1} + a_2b_{n/2+2} + \ldots a_{n/2}b_n) + \\
&x^{-2}(a_{n/2+1}b_1 + a_{n/2+2}b_2 + \ldots + a_nb_{n/2})
\end{aligned}$$

So from this we can see that $\mathbf{a}' \cdot \mathbf{b}'$ *contains* z, just as C' contains C. We simply need to add the cross terms (the last two lines in the above) into L and R respectively, since they share coefficients of x. Thus we redefine:

$$\begin{aligned}
L &= L_a + L_b + (a_1b_{n/2+1} + a_2b_{n/2+2} + \ldots a_{n/2}b_n)G \\
R &= R_a + R_b + (a_{n/2+1}b_1 + a_{n/2+2}b_2 + \ldots + a_nb_{n/2})G
\end{aligned}$$

and now the reduced commitment will satisfy:

$$C' = z'G + \mathbf{a}'\mathbf{G}' + \mathbf{b}'\mathbf{H}' = C + x^2 L + x^{-2} R, \quad z' = \mathbf{a}' \cdot \mathbf{b}'$$

as required.

So, much as previously mentioned, we can repeatedly apply this process: start with vectors length n, $C = zG + \mathbf{aG} + \mathbf{bH}$, pass this to the Verifier, along with L, R, receive back a challenge x, both sides recalculate C', continue until a final step (each step a halving and a new L, R), and in the last step reveal scalars for the now single values a, b, and the Verifier makes the final check that $C^* = a^*b^*G + a^*G_1 + b^*H_1$, where * indicates the $\log_2 n$-th transformed values.

As we mentioned at the start, to tweak this to create an argument of knowledge that a given z is the inner product of the committed vectors, "Protocol 1" requires the Verifier to provide an initial challenge x, and then the Prover runs the above algorithm replacing G with xG, so we have initially $C = z(xG) + \mathbf{aG} + \mathbf{bH}$.

6.1.3 Knowledge soundness

To get things absolutely clear, it'll help to draw out a table of the interaction between Prover and Verifier for a simple case; here let's say $n = 4$ so there are

only 2 reduction steps to the final reveal.

$C, n = 4$ (shared in advance)

	Prover		Verifier
(1)	L, R	$\rightarrow$	
		$\leftarrow$	x
	$[C']$		$[C']$
(2)	L', R'	$\rightarrow$	
		$\leftarrow$	x'
	$[C'']$		$[C'']$
(3)	a, b	$\rightarrow$	
			verify

First we observe that, by construction, every step is the same as every other; if we can demonstrate that we can open the original vectors $\mathbf{a}, \mathbf{b}$ given the intermediate transformed vectors $\mathbf{a}', \mathbf{b}'$ ((2)$\rightarrow$ (1)), it follows we could do the same going from (3) to (2) in the above; and indeed, at any intermediate step in a much longer list. So we will focus only on the former problem. Consider that we already have the values $\mathbf{a}', \mathbf{b}', z'$ generated in (2), for any x ; we'll write $\mathbf{a}'_i, \mathbf{b}'_i, z'_i$ for multiple values.

We'll now demonstrate that the Extractor can, using this, extract the original $\mathbf{a}', \mathbf{b}', z$ using rewinding.

Here we will need *three* transcripts, specifically, because there are three commitments C, L, R in the construction of C'. This will generate three equations at step 2 (i.e. after the provision of the first x): $C'_i = x_i^2 L + C + x_i^{-2} R \; \forall i \in 1..3$. Note here that C, L, R do not differ in the three versions of the equation, only C' does; L, R are calculated before provision of x. The Extractor can now simply solve this system of three equations in the three unknowns (details are obvious) to retrieve values of C, L, R in terms of the x_i, C'_is; but here we are only interested in C specifically. We have something like $C = f(C'_1, C'_2, C'_3)$ where f is *linear* function of those three variables. However, we also know the openings of those commitments! (Remember that we argued above that we can assume we **already know** $\mathbf{a}'_i, \mathbf{b}'_i, z'_i$, by recursion). For example, we know that $C'_1 = z'_1 G + \mathbf{a}'_1 \mathbf{G}'_1 + \mathbf{b}'_1 \mathbf{H}'_1$. Although $\mathbf{G}'_i, \mathbf{H}'_i$ are not the same as $\mathbf{G}, \mathbf{H}$, they are *linear* combinations of them, so opening is still provided in the original base. Hence you can see we are already done, for proving knowledge of opening, because you can write the opening of C as:

$$C = f((z'_1 G + \mathbf{a}'_1 \mathbf{G}'_1 + \mathbf{b}'_1 \mathbf{H}'_1), (z'_2 G + \mathbf{a}'_2 \mathbf{G}'_2 + \mathbf{b}'_2 \mathbf{H}'_2), (z'_3 G + \mathbf{a}'_3 \mathbf{G}'_3 + \mathbf{b}'_3 \mathbf{H}'_3))$$

Before we go on to complete the argument, remember the crucial point about binding in Pedersen commitments – any opening over the set of base points is **the** opening, so the above has already found $z, \mathbf{a}, \mathbf{b}$.

However, it does remain to show that the opening satisfies $z = \mathbf{a} \cdot \mathbf{b}$.

The opening of C given above will be of the form $z_C G + \mathbf{a}_C \mathbf{G} + \mathbf{b}_C \mathbf{H}$, where the subscripted variables are those extracted from the function f. Since we are here asserting that these values are indeed the correct openings, we could drop the C subscript, but we leave it in for clarity in the remaining steps.

We need to show that $z_C = \mathbf{a}_C \cdot \mathbf{b}_C$ to complete the proof. We already know, by the recursive argument, that $z' = \mathbf{a}' \cdot \mathbf{b}'$.

To do this, first observe that we can repeat the above procedure to solve for L and R just as we did for C. We can thus get openings $L = z_L G + \mathbf{a}_L \mathbf{G} + \mathbf{b}_L \mathbf{H}$ and $R = z_R G + \mathbf{a}_R \mathbf{G} + \mathbf{b}_R \mathbf{H}$. Then we can expand this into the original equation for C':

$$C' = \mathbf{a}' \cdot \mathbf{b}' G + \mathbf{a}' \mathbf{G}' + \mathbf{b}' \mathbf{H}' = (z_C + x^2 z_L + x^{-2} z_R) G + (\mathbf{a}_C + x^2 \mathbf{a}_L + x^{-2} \mathbf{a}_R) \mathbf{G} + (\mathbf{b}_C + x^2 \mathbf{b}_L + x^{-2} \mathbf{b}_R) \mathbf{H}$$

but note: we can do this *three* times with the randomly chosen three values x_1, x_2, x_3. This equation is therefore an **identity** over x. But the condition is even stronger: according to the binding property of such commitments it is also an "identity" over the individual base points. In particular the coefficient of G *and* x^0 must be equal; if we only consider the coefficient of (G, x^0), then we just need to look at the constant term in the dot product:

$$\begin{aligned} &\mathbf{a}' \cdot \mathbf{b}' = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n + \\ &x^2 (a_1 b_{n/2+1} + a_2 b_{n/2+2} + \ldots a_{n/2} b_n) + \\ &x^{-2} (a_{n/2+1} b_1 + a_{n/2+2} b_2 + \ldots + a_n b_{n/2}) \end{aligned}$$

which is indeed $\mathbf{a} \cdot \mathbf{b}$. Thus we have proved that $z_C = \mathbf{a} \cdot \mathbf{b}$ and we are done.

6.2 Encoding conditions into an inner product – a range proof

So far we have not attempted or considered applications of these techniques for compactly proving knowledge of vectors, or of the inner product of committed vectors. Only in this final section of the document will we do this – we'll specifically consider how Bulletproofs encodes a proof of the range of a committed number in an inner product, using polynomials (several, in fact!).

Start with a commitment to a number/value: $V = \gamma H + vG$, so a standard (hiding) Pedersen commitment to the value v with randomness γ. We want to prove that $v \in 0 \ldots 2^n - 1$.

This will map to Bitcoin's Confidential Transactions proposal of Maxwell et. al. [5], which uses Pedersen commitments to values in exactly this form, and leverages the commitment's homomorphism property to ensure balance.

6.2.1 Steps towards the range proof

Outline of the strategy

- The aim here is to achieve two goals: make the proof as compact as possible in terms of data communicated, but also to provide simultaneous proof of multiple conditions. To achieve the latter we leverage the idea outlined in 3.3.1, that is, using a challenge to evaluate a polynomial in order to fix it. This will be expanded on below. Note that this is done several times, in a layered approach, so *several* challenges and polynomials are involved. To achieve the former, we construct the outer polynomial such that one of its terms is an inner product, allowing us to leverage the very-compact inner product proof explained earlier in Section 6.

So, the following set of steps (actually, well explained in the paper!) is a way to encode this problem into an inner product verification of the type we saw in the last section.

Step 1: Encode the value v into a bit representation. Let $\mathbf{a}_L$ be a vector of bits such that $\mathbf{a}_L \cdot \mathbf{2}^n = v$ (put more simply, the components of $\mathbf{a}_L$ are the binary digits of v).

(Side note: from now we will use two additional pieces of notation: $\mathbf{k}^n = [1, k, k^2, \ldots, k^n]$ i.e. a vector of integer powers (but don't forget, all integers here are mod p), and $\mathbf{a} \circ \mathbf{b} = [a_1 b_1, a_2 b_2, \ldots, a_n b_n]$, often known as the "Hadamard product" of two vectors. Note that the Hadamard product is a vector, whereas the dot product/inner product is a scalar).

Step 2: In order to prove that v is in range, we'll combine this structure with an additional condition: that each element of $\mathbf{a}_L$ is either 0 or 1. To do that, we construct a "complementary" vector $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n$ and require that $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}$ hold.

We're set the task of constructing an algorithm for the Prover to prove to the Verifier that these conditions hold. To prove that these equations hold using an inner product, you could have the Verifier send a random x and then the Prover could prove e.g. $(\mathbf{a}_L \circ \mathbf{a}_R) \cdot \mathbf{x}^n = 0$; this would be correct in the "computational" sense (note: we argued in section 3.3.1 that this should be convincing with forgery probability negligible at $\simeq n/p$). But remember – the inner product argument of knowledge, constructed and proved in the previous two sections, was *not* zero-knowledge.

Step 3: The problems so far described are divided into effectively three stages. Here are the first two: first, for each of the conditions mentioned in Step 2 (vectors differ by 1, Hadamard is zero), use a challenge y to construct the inner product. Second, combine all three (including the condition on v) into a polynomial function of a second challenge z:

$$z^2 \mathbf{a}_L \cdot \mathbf{2}^n + z\,(\mathbf{a}_L - \mathbf{1}^n - \mathbf{a}_R) \cdot \mathbf{y}^n + (\mathbf{a}_L \circ \mathbf{a}_R) \cdot \mathbf{y}^n = z^2 v$$

Note that the use of z here is simply an "overlay" of another instance of the previous pattern/argument: use powers of a challenge to effectively evaluate a polynomial at that challenge. However, the y challenge required a power of n, i.e. n-th degree polynomials, because it is testing a condition on the set of components of the vectors (dimension n), whereas the z challenge only requires a quadratic polynomial, because we are only testing three conditions – those mentioned in Steps 1 and 2.

(Note that the second and third dot products are required to be zero, and therefore the RHS is just the first term).

Step 4: (Confusingly, this is not the third of the three stages I just mentioned :)).Using some deft mathematical footwork, convert this into a *single* inner product. We can afford for this factorization to leave terms "dangling", but what's important is that the $\mathbf{a}_L, \mathbf{a}_R$ terms be kept inside (since they can't be shared with the Verifier):

$$(\mathbf{a}_L - z\mathbf{1}^n) \cdot \left(\mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}^n) + z^2 \mathbf{2}^n\right) = z^2 v + \delta(y, z)$$

where, by expansion you can see that the "dangling" term δ is purely a function of the public challenges: $(z - z^2)(\mathbf{1}^n \cdot \mathbf{y}^n) - z^3(\mathbf{1}^n \cdot \mathbf{2}^n)$.

Step 5: The third of the three stages mentioned: To solve the problem that this is not zero knowledge, we need to blind the vectors $\mathbf{a}_L, \mathbf{a}_R$: introduce two new vectors $\mathbf{s}_L, \mathbf{s}_R$, created at random by the Prover. On creating these, the Prover can send commitments to these vectors; these are properly blinded vector Pedersen commitments:

$$A = \alpha H + \mathbf{a}_L \mathbf{G} + \mathbf{a}_R \mathbf{H}$$
$$S = \rho H + \mathbf{s}_L \mathbf{G} + \mathbf{s}_R \mathbf{H}$$

Step 6: On receipt of challenge values y, z, the Prover is now able to construct the above inner product. However the inner product needs to be embedded as the constant term in yet a *third* polynomial, using a new challenge x. This is to make the blinding effect of work. It's analogous to how in Section 4.2 we needed to construct $e\mathbf{x} + \mathbf{d}$ with blinding vectors $\mathbf{d}$, so that the private vectors $\mathbf{x}$ could be transferred. We thus *can* prove knowledge of $\mathbf{a}_L, \mathbf{a}_R$ without revealing them by transferring the vectors $\mathbf{l}, \mathbf{r}$ defined below(whether we actually will is another question!). The two terms of the dot product above are set as the constant term, while $\mathbf{s}_L, \mathbf{s}_R$ are the coefficient of x^1, in the following two linear polynomials, which are combined into a quadratic in x:

$$\mathbf{l}(x) = (\mathbf{a}_L - z\mathbf{1}^n) + \mathbf{s}_L x$$
$$\mathbf{r}(x) = \mathbf{y}^n \circ (\mathbf{a}_R + z\mathbf{1}^n + \mathbf{s}_R x) + z^2 \mathbf{2}^n$$
$$t(x) = \mathbf{l}(x) \cdot \mathbf{r}(x) = t_0 + t_1 x + t_2 x^2$$

Note that t is scalar-valued, not vector-valued, because it is the result of the inner product.

Let's recap the above 6 steps (they're not the full interactive protocol between Prover and Verifier, but they do cover the most important ideas behind it): we have the actual data representing the value in $\mathbf{a}_L, \mathbf{a}_R$. We use the first challenge, y, to "fix" the bits of those vectors, then use the challenge z to assert the three constraints (all of which are inner products) in one equation. Then we combine the three terms into one inner product, with dangling terms OK because they are using public data. Finally we use corresponding random vectors $\mathbf{s}_L, \mathbf{s}_R$ to the original vectors, and require a third challenge to make a quadratic polynomial in *that* challenge for which the constant term is the inner product we've constructed. Verifying the value of that inner product will validate that the value v is in range, while because it has been encoded as an inner product, we can leverage the above inner product proof for the desired compactness (this will be explained in 6.2.3).

6.2.2 After having built the "outer" polynomial t(x)

Here we'll briefly describe what the Prover needs to do to convince the Verifier that his value is in range, ignoring the inner product proof. Then we'll overlay that in the succeeding section.

The Prover will, basically, have to convince the verifier that he has honestly constructed the polynomial $t(x)$. To do this he sends Pedersen commitments T_1, T_2 $(= \tau_{1,2} H + t_{1,2} G)$ to the coefficients t_1, t_2, obviously *before* he receives the

final challenge x; after having received that, he then can send the "blinding" half of the T commitments (τ), combined into the form required to make the "committed" version of the x-polynomial add up (this will be $\tau_x = \tau_2 x^2 + \tau_1 x + z^2\gamma$). He'll also send the dot product $t(x) = \mathbf{l}(x) \cdot \mathbf{r}(x) = \hat{t}$, and the blinding factor required for the Verifier to verify the earlier commitments A, S: : $\mu = \alpha + \rho x$. The check that the Pedersen commitment to the value (V) is indeed the constant term of $t(x)$ will thus look like this:

$$\hat{t}G + \tau_x H =? \quad z^2 V + \delta G + xT_1 + x^2 T_2$$

where you can verify that both sides of the equation are the "Pedersen-ised" form of:

$$\begin{aligned} &\mathbf{l}(x) \cdot \mathbf{r}(x) = z^2 v + \delta + xt_1 + x^2 t_2 \quad \text{for } G \\ &\tau_2 x^2 + \tau_1 x + z^2 \gamma \quad \text{for } H \end{aligned}$$

The verifier must also check that the commitments A, S are valid. In other words, he must check that the blinded commitments to $\mathbf{a}_L, \mathbf{a}_R$ that the Prover made before receiving the challenges x, y, z match the constructions of $\mathbf{l}, \mathbf{r}$. Refer back to the definitions of those two vectors in terms of $\mathbf{a}_L, \mathbf{a}_R$, and you can see that this requires:

$$\begin{aligned} &\mathbf{H}' = \mathbf{y}^{-n}\mathbf{H} \\ &P = A + xS - zG + \left(z\mathbf{y}^n + z^2\mathbf{2}^n\right)\mathbf{H}' \\ &P =? \ \mu H + \mathbf{l}\mathbf{G} + \mathbf{r}\mathbf{H}' \end{aligned}$$

The introduction of $\mathbf{H}'$ is a bit of algebraic housekeeping to relate the commitment to $\mathbf{a}_R$ in A to the definition of $\mathbf{r}(x)$, which uses a Hadamard product with $\mathbf{y}^n$; note it has nothing to do with the use of ' notation in the inner product proof.

The Verifier must also, finally, check that the inner product is correct: $\hat{t} =? \ \mathbf{l} \cdot \mathbf{r}$.

Let's compile the interaction between Prover and Verifier for what we have so far:

$V, n, \mathbf{G}, \mathbf{H}, G, H$ (shared in advance)

Prover		Verifier
A, S	$\rightarrow$	
	$\leftarrow$	y, z
T_1, T_2	$\rightarrow$	
	$\leftarrow$	x
$\tau_x, \mu, \hat{t}, \mathbf{l}, \mathbf{r}$	$\rightarrow$	
		verify

, where "verify" refers to the three checks marked =? above.This provides the necessary zero-knowledgeness because $\mathbf{l}, \mathbf{r}$ are properly blinded as already explained, but it doesn't provide the compactness; see next section.

6.2.3 Leveraging the compact O(logn) inner product proof

As a reminder the compact inner product proof provided in the Bulletproofs paper takes as *public* input only the commitment C and the claimed inner product (which for continuity I have called z, but in the paper c is used); assuming that everything else is set in advance – the basepoints $G, H, \mathbf{G}, \mathbf{H}$, and the dimension n. We can convert the final check $\hat{t} =? \mathbf{l} \cdot \mathbf{r}$ above, with the check of the commitment P above, into one check of such an inner product proof; we just need to set the basepoints to make it fit:

public:
$P - \mu H \rightarrow C$
$\mathbf{H}' = \mathbf{y}^{-n}\mathbf{H} \rightarrow \mathbf{H}$
$\mathbf{G} \rightarrow \mathbf{G}$
$\hat{t} \rightarrow z$
private:
$\mathbf{l}(x) \rightarrow \mathbf{a}$
$\mathbf{r}(x) \rightarrow \mathbf{b}$

Given these renamings, we can apply the inner product proof of Section 6.1 directly.

How does it change the interaction? Instead of having to transmit $\mathbf{l}, \mathbf{r}$ (and don't be deceived by the fact that's only 2 terms – they are vectors, so by far the largest part of the transfer in the existing protocol; also, don't be deceived thinking "oh but they're bit decompositions – each element is 0 or 1!" - this is wrong because we blinded them, so they're all group elements, which take 32 bytes each for a 256 bit curve, so for a 32 bit number you need 32x32x2 bytes), the Prover can instead transfer only $P - \mu H$ as C. He transfers $\hat{t}$ in either case, so no change there. Note that as well as the inner product proof, the other pre-existing check must still be carried out by the Verifier:

$$\hat{t}G + \tau_x H =? \quad z^2 V + \delta G + xT_1 + x^2 T_2$$

(this is what binds the proof to the actual original Pedersen commitment V to v).

At this point, we are done: we have a proof which is claimed to be sound (see 6.2.6), and zero knowledge (see 6.2.7), and which is also very compact (see 6.2.5). The proof will be constructed non-interactively (see 6.2.4). We claim that all the necessary conditions hold, to prove that V is a commitment to a value v that is in the range $0 \dots 2^n - 1$.

6.2.4 Aside: Non-interactive proofs

The above protocol is obviously pretty complicated, but more importantly, it has several steps of interaction between Prover and Verifier. This is not acceptable for the application of Bitcoin transactions; but it can be solved with a standard technique that's used widely in all kinds of cryptographic protocols that involve this kind of interaction. It's called the Fiat-Shamir heuristic [22]. This point

is explained in Section 4.4 of the Bulletproofs paper. It also applies to most of the other constructions we've looked at in this document – for example the Schnorr signature, where we can replace the interactive challenge e with a hash of (message to be signed, nonce point R, public key P).

Using this heuristic however does come at a cost – it relies on something called the "Random Oracle Model"(discussed at length in [23] - warning, it's a rabbit hole!) or "ROM" for short, in other words, it's another assumption you have to make in order to reach the conclusion that the protocol is secure. Basically a "random oracle" is considered to be a black box that outputs unpredictable, random values in response to input, in a deterministic way (that is to say, if you give it the same input twice, it will give the same random output).

Using the ROM is a tradeoff which people do, generally, make since non-interactivity makes a lot of things that would otherwise be impractical, practical.

It should also be mentioned that, to address how the ROM changes the zero knowledge proofs we have discussed thus far is a separate and interesting topic. Basically we have to treat the random oracle as something that the Extractor or Simulator can program in advance.

It's also important to note that the input to the random oracle, in the Fiat-Shamir heuristic, is specifically **the transcript of the interaction up to that point**.

6.2.5 Scaling

Here we'll focus (as in previous sections) *only* on the amount of data communicated; not the computational cost, although of course this is indeed very important. We assume use of the Fiat-Shamir heuristic just mentioned; this means that we get the "challenge" values (for example, x, y, z, but also the challenges in the inner product proof) for "free" since they're calculated by the Prover alone, using a cryptographic hash function like SHA256.

The Prover must send the curve points A, S, T_1, T_2, and also the scalar values τ_x, μ and $\hat{t}$, but then also L, R the pairs (of which there are $2\log_2 n$) along with the final 2 scalars a, b at the end of that inner product protocol. So the total size of the published proof is (curve point size)*$(4 + 2\log_2(n))$ + (scalar size)*5.

Both curve points and scalars in the group can be encoded in about 32 bytes, so this is *roughly* $32 \times (9 + 2\log_2(n))$, where n is the number of bits in the range. For 32 bit ranges this gives around 620 bytes, and for 64 bits, it's nearer to 700 bytes. Note that this is **dramatically smaller** than the range proofs based on Borromean ring signatures[12] for the earlier implementation of Confidential Transactions, which used about 2500 bytes in its simplest form for 32 bit ranges, and would be *far* higher for 64 bits (because that construction didn't have log(n) scaling)!

6.2.6 Knowledge soundness

This is obviously a slightly more complicated proof than previous ones, because the algorithm itself has several components, also my exposition will be a little lacking in detail.

Fortunately, though, it's mostly using logic very similar to previous proofs, so it hopefully isn't too overwhelming.

The proof given in Appendix C of the Bulletproofs paper is for the *aggregated* form of the proofs (see 6.2.8), which makes sense as a single proof is just a special case of the aggregated form.

Outline Start by noting that Section 6.1.3 gave us a proof of soundness of the inner product proof (i.e. the soundness of an argument of knowledge for the vectors $\mathbf{a}, \mathbf{b}$ such that their dot product is z). We of course use this here; we assert that this provides us with a sound opening of the vectors $\mathbf{l}, \mathbf{r}$, and asserts that their inner product is $\hat{t}$ (see the table in 6.2.3 for the conversion of terms).

As usual in such proofs, we use an Extractor who can rewind at any point in the conversation. Here, for the first part of the proof, we will specifically be rewinding the x-challenge, but not the previous two challenges y, z. That means that in the following equations, the values y, z are fixed, but there is a different x value for each generated proof/equation. For the second part of the proof, we will also need $n + 2$ different y, z challenges.

We proceed as follows: starting with the formula for the commitment P, we can run this twice and get openings first of the quantities: $\alpha, \rho, \mathbf{a}_L, \mathbf{a}_R, \mathbf{s}_L, \mathbf{s}_R$. We can then use the public value $\hat{t}$ along with three transcripts (i.e. three x values), and get openings of the quantities $t_1, t_2, \tau_1, \tau_2, v, \gamma$. At this point we have extracted the value under commitment/range proof: v.

However, we must also proof that the conditions hold, that is: $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}, \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n, \mathbf{a}_L \cdot \mathbf{2}^n = v$. By using here different values of the challenges y, z, and comparing coefficients in $t(x) = \mathbf{l}(x) \cdot \mathbf{r}(x)$ we can justify that $t_0 = z^2 v + \delta$ has the right form, and thus that these 3 conditions are true.

Stage 1: Openings of $\alpha, \rho, \mathbf{a}_L, \mathbf{a}_R, \mathbf{s}_L, \mathbf{s}_R$

Remembering that P is defined as $\mu H + \mathbf{l}\mathbf{G} + \mathbf{r}\mathbf{H}'$, and noting its construction by the verifier as $P = A + xS - zG + \left(z\mathbf{y}^n + z^2\mathbf{2}^n\right)\mathbf{H}'$, we will compare the two. For the first challenge x_1, this will look like:

$$P_1 = \mu_1 H + \mathbf{l}_1\mathbf{G} + \mathbf{r}_1\mathbf{H}' = A + x_1 S - z\mathbf{G} + (z\mathbf{y}^n + z^2\mathbf{2}^n)\mathbf{H}'$$

Do likewise for x_2. Now consider the "coefficients" (recall the idea here in Section 6.1.3) of H:

$$\begin{aligned} \mu_1 &= \alpha + \rho x_1 \\ \mu_2 &= \alpha + \rho x_2 \\ \text{solve:} &\quad \rho, \alpha \end{aligned}$$

Then consider coefficients of $\mathbf{G}$:

$$\begin{aligned} \mathbf{l}_1 &= \mathbf{a}_L + x_1 \mathbf{s}_L - z \\ \mathbf{l}_2 &= \mathbf{a}_L + x_2 \mathbf{s}_L - z \\ \text{solve:} &\quad \mathbf{a}_L, \mathbf{s}_L \end{aligned}$$

And finally for $\mathbf{H}'$ (note that these curve points are fixed for fixed y, as here):

$$\begin{aligned}
&\mathbf{l}_1 = \mathbf{a}_R + x_1\mathbf{s}_R + k \\
&\mathbf{l}_2 = \mathbf{a}_R + x_2\mathbf{s}_R + k \\
&\text{solve:} \quad \mathbf{a}_R, \mathbf{s}_R
\end{aligned}$$

where k is just the remaining terms for $\mathbf{H}'$, and is constant.

Now we have explicit openings for $\alpha, \rho, \mathbf{a}_L, \mathbf{a}_R, \mathbf{s}_L, \mathbf{s}_R$.

Stage 2: Openings of $t_1, t_2, \tau_1, \tau_2, v, \gamma$

The model is basically the same as above; here we actually need 3 values x_1, x_2, x_3. We consider here the Verifier's check that $\hat{t}G + \tau_x H = z^2 V + \delta(y,z)G + xT_1 + x^2 T_2$. Recall first that $T_{1,2} = \tau_{1,2}H + t_{1,2}G$, that $V = \gamma H + vG$, and that δ is a publically known constant (once y and z are fixed, as here).

Applying the three challenges and considering the G coefficient gives:

$$\begin{aligned}
&\hat{t}_1 = z^2 v + \delta + x_1 t_1 + x_1^2 t_2 \\
&\hat{t}_2 = z^2 v + \delta + x_2 t_1 + x_2^2 t_2 \\
&\hat{t}_3 = z^2 v + \delta + x_3 t_1 + x_3^2 t_2 \\
&\text{solve:} \quad t_1, t_2, v
\end{aligned}$$

At this point we have extracted the value v.

Considering the H coefficient gives:

$$\begin{aligned}
&\tau_{x1} = z^2 \gamma + x_1 \tau_1 + x_1^2 \tau_2 \\
&\tau_{x2} = z^2 \gamma + x_2 \tau_1 + x_2^2 \tau_2 \\
&\tau_{x3} = z^2 \gamma + x_3 \tau_1 + x_3^2 \tau_2 \\
&\text{solve:} \quad \tau_1, \tau_2, \gamma
\end{aligned}$$

Now we have explicit openings for $t_1, t_2, \tau_1, \tau_2, v, \gamma$.

Stage 3: It remains to prove that the 3 conditions hold: $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}, \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n, \mathbf{a}_L \cdot \mathbf{2}^n = v$.

For this part, we need that the Extractor runs the Prover with n different y challenges (and this is applied multiplicatively with the 3 different x challenges we have already mentioned).

We must argue that $t(x) = \mathbf{l}(x) \cdot \mathbf{r}(x)$ is an identity over x. We have already extracted t_0, t_1 and t_2 (with $t_0 = \delta(y,z) + z^2 v$), so we have fully fixed $t(x)$ for any particular y, z. On the other hand, $\mathbf{l}(x) \cdot \mathbf{r}(x)$ is fixed from the first step, by our re-use of the opening of those vectors from 6.1.3 (see the start of this section). Let $p(x) = \mathbf{l}(x) \cdot \mathbf{r}(x)$. We can now check whether $t(x) - p(x) = 0$ holds for each of our 3 challenges x_1, x_2, x_3. If it does, we assert that the equation is an identity, because the polynomial has degree 2 over x, which is less than 3. This specifically means that $p_0 = t_0$ and we can use the expansion of $\mathbf{l}, \mathbf{r}$ to assert that:

$$z^2 v + \delta(y,z) = z^2(\mathbf{a}_L \cdot \mathbf{2}^n) + \mathbf{y}^n \cdot (\mathbf{a}_L \circ \mathbf{a}_R) + z((\mathbf{a}_L - \mathbf{a}_R - \mathbf{1}^n) \cdot \mathbf{y}^n) + \delta(y,z)$$

We cancel the δ terms as they are identical in every run. This polynomial has degree $n-1$ over y (remember, a term like $\mathbf{v} \cdot \mathbf{y}^n$ is $v_0 + v_1 y + v_2 y^2 +$

$\ldots v_{n-1}y^{n-1}$), and we would therefore require n versions of that challenge to determine the system of linear equations and extract a solution (in other words, fix the coefficients of that polynomial). However it is also a quadratic in z, so overall it is a bivariate polynomial of degree $n-1+2$, requiring $n+2$ combinations to fix its coefficients. After this, we can deduce that coefficients of the same powers of $(y, z$ must be equal. In particular, the coefficient of $y^0, z^2)$ must be equal:

$$v = \mathbf{a}_L \cdot \mathbf{2}^n$$

And the coefficient of (y^0, z^1) (zero on the LHS) must be equal:

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}$$

Likewise the coefficient of (y^0, z^0):

$$\mathbf{a}_L - \mathbf{a}_R - \mathbf{1}^n = \mathbf{0}$$

In other words, all of our three conditions hold: $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}, \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n, \mathbf{a}_L \cdot \mathbf{2}^n = v$. This ends the proof.

6.2.7 Zero-Knowledgeness

Here is a rough outline, only. The usual trick, if you recall from earlier sections, is to examine the transcript for the interactive version of the protocol and deduce how it could be created without knowing the secret data (the witness), if necessary allowing a Simulator entity to rewind protocol execution.

The witness here is specifically the values v, γ such that $V = vG + \gamma H$. V is treated as the public input to the protocol. The transcripts look like $((A, S), y, z, (T_1, T_2), x, (\hat{t}, \mu, \tau_x))$, where the Prover's part of the conversation is parenthesized.

As usual, what you need to do is to pick values at random except have certain values be calculable from the other random values, so that the verification conditions hold. As a reminder, the verifications, ignoring the inner product proof were:

$$\begin{aligned}
&\hat{t}G + \tau_x H =? \quad z^2 V + \delta(y, z)G + xT_1 + x^2 T_2 \\
&A + xS - zG + \left(z\mathbf{y}^n + z^2\mathbf{2}^n\right)\mathbf{H}' =? \ \mu H + \mathbf{lG} + \mathbf{rH}'
\end{aligned}$$

Remember that the public commitment V is only checked in the first of the above two equations; so forging that equation successfully is the main goal. To get that equation to verify, given random $T_2, \tau_x, z, x, y, \hat{t}$ we can simply calculate T_1 as:

$$T_1 = x^{-1}\left((\hat{t} - \delta(y, z))G - z^2 V - x^2 T_2 + \tau_x H\right)$$

To make the second equation verify, the one that checks the validity of the commitment P, we can choose random $A, \mu, \mathbf{l}, \mathbf{r}$ in addition to the other random values already mentioned, and ensure that $\hat{t} = \mathbf{l} \cdot \mathbf{r}$, and set the commitment S to value:

$$S = x^{-1}\left(\mu H + zG + \mathbf{lG} + (\mathbf{r} - z\mathbf{y}^n - z^2\mathbf{2}^n)\mathbf{H}' - A\right)$$

Then the transcript $((A, S^*), y, z, (T_1^*, T_2), x, (\hat{t}, \mu, \tau_x))$ will verify, where we use * to indicate calculated values, and all other values are randomly selected, if we create additional random vectors $\mathbf{l}, \mathbf{r}$ such that their dot product is $\hat{t}$.

Based on the usual reasoning, the ability to create fake transcripts which verify implies that the protocol releases no information other than that the witness is valid. Caveat: this "proof" is, again, a brief sketch, and more detail is really needed.

6.2.8 Aggregation

This is just a brief overview, although the core idea is pretty simple.

The motivation here is to leverage the O(logn) scaling achieved in the inner product proof. It turns out that you can construct a single proof for the range of *multiple* values v, while only incurring an additional space cost of $2\log_2(m)$ for m additional values v – which is remarkably useful, for example in aggregating multiple outputs in a Bitcoin transactions. Note how this (along with batching the verification of proofs in transactions, something we haven't discussed at all here), could actually incentivize creating transactions with larger numbers of outputs – see e.g. Coinjoin.

To prove that multiple values v_j, each of which has a commitment V_j, are in range, we can do something quite crude – just concatenate the values together. For example, if the first value is 10 and the second value is 3, you could write, using $n = 4, v_1 = 1010, v_2 = 0011$, and concatenate to 10100011 in 8 bits instead of 4. This is the fundamental way we achieve the space saving mentioned above. Consider that you had two values v_1, v_2, and let's assume 64 bit range proofs. According to the formula from 6.2.4, the total size of 2 proofs is $2 \times (32 \times (9 + 2\log_2(64))) = 1344$ bytes. If instead we concatenate, we have effectively a 128 bit range proof for a single number, giving $32 \times (9 + 2\log_2(128)) = 736$ bytes. The formulae show clearly the idea – move the multiplication from multiple proofs inside the log term.

However, as explained in the paper (section 4.3), this needs some "patching up" to make the full rangeproof work. The values δ, τ_x, and the commitment P must be updated, to reflect the existence of multiple values, also. I leave the details to the paper for those interested.

6.3 General arithmetic circuits

Giving some detail on this is deferred to a later version of this document. Here will just be a few notes.

It's not hard, having gone through the details of the above inner product proof → range proof, to imagine that one could make different types of zero knowledge proofs using the same equipment; since ultimately we just kind of "hacked-in" the set of constraints: $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}, \mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n, \mathbf{a}_L \cdot \mathbf{2}^n = v$ on a pair of starting vectors that were introduced at the start.

The concept of an arithmetic circuit can be found in [24]. These circuits can be thought of as encodings of polynomials, where at each point in the circuit, a value is found by either adding or multiplying two input gates. The Bulletproofs paper, in Section 5, discusses the case of a Hadamard product of the type we've already encountered, which is really a set of multiplications of pairs of values,

i.e.

$$\mathbf{a}_L \circ \mathbf{a}_R = (a_{L1} \times a_{R1}, a_{L2} \times a_{R2}, \ldots, a_{Ln} \times a_{Rn}) = \mathbf{a}_O$$

Using a result from Bootle [2] not here discussed, it's possible to combine an assertion about a Hadamard product as above with a set of linear constraints on other vectors $\mathbf{w}_{Lq}, \mathbf{w}_{Rq}, \mathbf{w}_{Oq}$, where there are $Q \leq 2n$ copies of these, and to get from this an arbitrary arithmetic circuit.

The paper then constructs a proof where we are proving that $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$ (c.f. the previous $\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{0}$, which is obviously easier), and that the above mentioned linear constraints hold (they are expressed as dot products). The protocol is obviously more complex than the range proof one, but preserves the same basic structure. The initial commitment A is replaced with two commitments A_I to $\mathbf{a}_L, \mathbf{a}_R$ and A_O to $\mathbf{a}_O$ and requires more commitments T_i to the coefficients in the more complex version of the polynomial $t(x)$ (but still uses the same three challenges x, y, z). See Protocol 3 in the paper for more.

There are a wide variety of potential applications of such a construction, as I understand, and in a sense this document "buries the lede", since some of these applications may end up being more practically useful than the presented range proof. To quote from [25]:

> As a specific example, consider a single run of the SHA2 compression function. Our prover requires less than 30 MiB of memory and about 21 seconds to prove knowledge of a SHA2 preimage. Verification takes about 23 MiB of memory and 75 ms, but we can batch verify additional proofs in about 5 ms and 13.4 KiB each.

This is just one (famously difficult) case: proof of knowledge of a hash preimage. There may other variants of the same idea (hashes based on Pedersen commitments?) that end up being more practical. The key takeaway here is that this a zero knowledge proof system for *arbitrary* computation, whose performance is notably better than alternatives in some scenarios, and whose security is only dependent on the well studied ECDLP problem.

6.4 Implementation

Implementation in libsecp256k1, the secp256k1 elliptic curve code implementation used by Bitcoin, is in [13], and is primarily the work of Andrew Poelstra up till now. Additional existing partial implementations are those in Monero [26] and an initial proof of concept by Bünz [27].

6.4.1 Boosting verification performance

In previous sections we've focused exclusively on the *size* of the proofs being generated, with the "headline" result of Bulletproofs and the earlier Bootle construction being that the size of the proof is *logarithmic* in , the size of the vectors.

However other performance characteristics matter a lot too, depending on context, for example: memory and time required for proof construction, and memory and time required for verification. The latter is particularly important in a Bitcoin or other cryptocurrency scenario, since all participants must verify.

A big speedup can be achieved by using a technique usually called "multi-exponentiation" (confusingly: in an elliptic curve context we're actually talking

about multi-multiplication e.g. $aG_1 + bG_2 + cG_3 + \ldots$; but the term "exponentiation" is used because in the discrete log case this is actually exponentiation: $g_1^a g_2^b g_3^b \ldots$). It's not obvious why it should be faster to compute the sum of a bunch of scalar multiples like this; one would expect it would require $N\times$ a single scalar multiple computation, but it turns out that with clever algorithms, such as Bos-Coster [28] or Pippenger (29), the scaling can be reduced to $O(\frac{n}{\log n})$.

The above refers to how verification of a single proof can be sped up. However, further performance improvements can be achieved by **batching** multiple verifications of multiple proofs together. Combining two "exponentiations" (or "multiexponentiations") together involves using a similar trick to that employed several times in this document, that is to say combining two values by evaluating a linear polynomial with a random input α, i.e. $\alpha x + y$. For more information see Section 6.2 of Bulletproofs.

7 References

1. Groth 2009 `http://www.cs.ucl.ac.uk/staff/J.Groth/MatrixZK.pdf`
2. Bootle et al. 2016 `https://eprint.iacr.org/2016/263`
3. Lindell 2003: `https://eprint.iacr.org/2001/107` (updated in 2003 with the section on Witness Extraction, 3.3)
4. Bootle's without-maths explanation of the inner product proof innovation 2016 `https://www.benthamsgaze.org/2016/10/25/how-to-do-zero-knowledge-from-di`
5. Maxwell 2015 `https://www.elementsproject.org/elements/confidential-transactions/investigation.html`
6. My deep dive into (original flavour) Confidential Transactions 2015 `https://github.com/AdamISZ/ConfidentialTransactionsDoc`
7. Rosenberg, A more digestible explanation of the core crypto concepts in Confidential Transactions 2017 `http://cryptoservices.github.io/cryptography/2017/07/21/Sigs.html`
8. Wikipedia, "Nothing Up My Sleeve numbers" `https://en.wikipedia.org/wiki/Nothing_up_my_sleeve_number`
9. Wikipedia, "ElGamal encryption scheme" (can be used as a commitment scheme) `https://en.wikipedia.org/wiki/ElGamal_encryption`
10. Ruffing and Malavolta 2017 `https://eprint.iacr.org/2017/237.pdf`
11. Wikipedia, random oracles (and the Random Oracle Model) `https://en.wikipedia.org/wiki/Random_oracle`
12. Maxwell and Poelstra 2015 Borromean Ring Signatures `https://github.com/Blockstream/borromean_paper/blob/master/borromean.pdf`
13. Poelstra's implementation of Bulletproofs in libsecp256k1 (WIP) 2018 `https://github.com/ElementsProject/secp256k1-zkp/pull/16`
14. Green, blog posts on ZKPs 2014++ `https://blog.cryptographyengineering.com/2014/11/27/zero-knowledge-proofs-illustrated-primer/`
15. Bünz et al. "Bulletproofs" 2017 `https://eprint.iacr.org/2017/1066`
16. Wikipedia, the One Time Pad `https://en.wikipedia.org/wiki/One-time_pad`
17. Boneh and Shoup (WIP) 2017 `https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf` . See Chapter 19,20 for the development of the ideas behind Zero Knowledge Proofs.
18. Wikipedia, Zero Knowledge Proofs `https://en.wikipedia.org/wiki/`

Zero-knowledge_proof
19. Wikipedia, the Vandermonde matrix https://en.wikipedia.org/wiki/Vandermonde_matrix
20. Goldwasser, Micali, Rackoff 1985 https://groups.csail.mit.edu/cis/pubs/shafi/1985-stoc.pdf
21. Groth, Bayer 2012 http://www.cs.ucl.ac.uk/staff/J.Groth/MinimalShuffle.pdf
22. Wikipedia, the Fiat-Shamir heuristic https://en.wikipedia.org/wiki/Fiat%E2%80%93Shamir_heuristic
23. Green, blog posts on the Random Oracle Model 2011 https://blog.cryptographyengineering.com/2011/09/29/what-is-random-oracle-model-and-why-3/
24. Wikipedia, Arithmetic circuits https://en.wikipedia.org/wiki/Arithmetic_circuit_complexity
25. Poelstra blog post on Bulletproofs 2018 https://blockstream.com/2018/02/21/bulletproofs-faster-rangeproofs-and-much-more.html
26. Monero implementation of bulletproofs (WIP) 2018 https://github.com/moneromooo-monero/bitmonero/tree/bp-multi-aggregation
27. Bünz's proof of concept implementation of Bulletproofs in Java 2017 https://github.com/bbuenz/BulletProofLib
28. Bos, Coster 1990 https://link.springer.com/content/pdf/10.1007/0-387-34805-0_37.pdf
29. Bernstein et. al. 2012 http://eprint.iacr.org/2012/549.pdf (see page 15)