18

SNARK Composition and Recursion

18.1 Composing Two Different SNARKs

Consider two SNARK systems, $\mathcal{I}$ and $\mathcal{O}$, say for arithmetic circuit-satisfiability, with different cost profiles. The prover in $\mathcal{I}$ is very fast (say, linear in the size of the statement being proven), but the proofs and verification time are fairly large (though still sublinear in the size of the statement being proven, e.g., square root of the circuit size). In contrast, the prover in $\mathcal{O}$ is slower—say, superlinear in the size of the circuit by logarithmic factors, and with a large leading constant factor—but the proofs and verification time are very short and fast (say, of length logarithmic or even constant in the circuit size). Is it possible to combine them to get the best of both worlds? That is, we seek a SNARK $\mathcal{F}$ with the fast prover speed of $\mathcal{I}$ and the short proof length and fast verification of $\mathcal{O}$.

The answer is yes, at least in principle, via a technique called proof composition. This works as follows. Suppose the $\mathcal{F}$-prover $\mathcal{P}_{\mathcal{F}}$ claims to know a witness w such that $\mathcal{C}(w) = 1$, where $\mathcal{C}$ is a specified circuit. $\mathcal{P}_{\mathcal{F}}$ can use $\mathcal{I}$ to generate a SNARK proof π of the claim at hand. But since π is pretty big and verifying it is somewhat slow, $\mathcal{P}_{\mathcal{F}}$ doesn't want to explicitly send π to the $\mathcal{F}$ verifier. Rather, $\mathcal{P}_{\mathcal{F}}$ can use the $\mathcal{O}$-SNARK

system to *prove* to the $\mathcal{F}$-verifier that it knows π. It is this $\mathcal{O}$-proof π' that $\mathcal{P}_{\mathcal{F}}$ actually sends to the verifier. Put another way, $\mathcal{P}_{\mathcal{F}}$ uses the fast-verification SNARK $\mathcal{O}$ to establish knowledge of an $\mathcal{I}$-proof π that would have convinced the $\mathcal{I}$ verifier that $\mathcal{P}_{\mathcal{F}}$ knows a w such that $\mathcal{C}(w) = 1$.

The above procedure requires taking the verification procedure of $\mathcal{I}$ and feeding it through the proof machinery of $\mathcal{O}$. That is, the $\mathcal{I}$-verifier must be represented as an arithmetic circuit $\mathcal{C}'$ and the $\mathcal{O}$ prover then applied to $\mathcal{C}'$ to establish knowledge of a π such that $\mathcal{C}'(\pi) = 1$.[1]

Let $\mathcal{F} = \mathcal{O} \circ \mathcal{I}$ denote the above composed proof system. Here, $\mathcal{O}$ stands for the "outer" SNARK and $\mathcal{I}$ stands for the "inner" SNARK. The motivation for this terminology is that one thinks of the $\mathcal{O}$-proof π' that is actually sent to the verifier in $\mathcal{F}$ as having an $\mathcal{I}$-proof π "living inside of it": the $\mathcal{O}$-proof π' attests that whoever generated the proof knows some $\mathcal{I}$-proof π for the claim at hand.

Costs of the composed proof system. The final proof length and verification time of $\mathcal{F}$ is the size of the proof generated by $\mathcal{O}$ applied to the $\mathcal{I}$ verifier circuit $\mathcal{C}'$. Since the $\mathcal{O}$-proof and verification procedure are respectively short and fast, the $\mathcal{F}$-proof and verification procedure are short and fast as well.

The $\mathcal{F}$ prover first has to generate the $\mathcal{I}$-proof π for $\mathcal{C}$ (which is by assumption fast), and then has to generate the $\mathcal{O}$-proof for $\mathcal{C}'$. While the $\mathcal{O}$ prover is slow, the key point is that $\mathcal{C}'$ should be *much smaller* than $\mathcal{C}$, since the verification procedure of $\mathcal{I}$ is sublinear (e.g., square root) in the size of $\mathcal{C}$. Hence, the time required by the $\mathcal{F}$ prover to generate the $\mathcal{I}$-proof that $\mathcal{C}'(\pi) = 1$ should be dwarfed by the time required to compute π in the first place. Hence, the $\mathcal{F}$ prover time is extremely close to that of the $\mathcal{I}$ prover, which by assumption is fast. The best of both worlds has been achieved.

[1] There is nothing special about circuit-satisfiability in this example. What matters is that the verification procedure of $\mathcal{I}$ be represented in whatever format $\mathcal{O}$ requires to allow $\mathcal{P}$ to establish that it knows an $\mathcal{I}$-proof π that would have caused the $\mathcal{I}$ verifier to accept. See Section 6 and Section 8.4 for additional discussion of intermediate representations other than circuits, including R1CS.

There are other potential benefits of proof composition beyond reducing verification costs. For example, if the inner SNARK $\mathcal{I}$ is not zero-knowledge, but the outer SNARK $\mathcal{O}$ is zero-knowledge, the composed SNARK $\mathcal{F}$ will be zero-knowledge. Hence, composition can be used to transform a highly efficient but non-zero-knowledge SNARK $\mathcal{I}$ into a new SNARK $\mathcal{O} \circ \mathcal{I}$ that is zero-knowledge.

18.2 Deeper Compositions of SNARKs

As in the previous section, imagine a SNARK $\mathcal{I}$ for circuit satisfiability instances of size S in which the verification procedure, when itself represented as an arithmetic circuit, has size $O(S^{1/2})$, and proofs have size $O(S^{1/2})$ as well. That is, verification is sublinear relative to the cost of evaluating the circuit gate-by-gate on a witness w, but is still more expensive than we might like. In principle, self-composition can be used to obtain a SNARK with lower verification cost.

Composing $\mathcal{I}$ with *itself* yields a new SNARK $\mathcal{F} = \mathcal{I} \circ \mathcal{I}$ with proof size and verification time $O\left((S^{1/2})^{1/2}\right) = O\left(S^{1/4}\right)$. One more invocation of composition, say with $\mathcal{F}$ as the outer SNARK and with $\mathcal{I}$ as the inner SNARK, yields yet another SNARK, now with verification time $O\left((S^{1/4})^{1/2}\right) = O(S^{1/8})$. In this way, the more invocations of composition, the smaller the proofs and faster the verification time of the resulting SNARK. One can fruitfully continue this process until the verification circuit of the composed SNARK is smaller than the so-called *recursion threshold* of the base SNARK $\mathcal{I}$. This refers to the smallest circuit size S^* such that the verification procedure of $\mathcal{I}$ cannot be represented by a circuit-satisfiability instance of size smaller than S^*. On circuits smaller than the recursion threshold, composing the SNARK with itself does not reduce verification costs, and in fact may increase them.

Of course, the deeper the recursion, the more work the prover has to do. For example, if $\mathcal{I}$ is composed with itself three times, then the prover has to "in its own head" first produce a proof π that would convince the $\mathcal{I}$ verifier of the claim at hand, then produce a proof π' that it knows π,

then produce a proof π'' that it knows π'.[2] This is naturally more work than just producing the proof π for the non-composed proof system.

Establishing knowledge-soundness of composed SNARKs. When considering the composition $\mathcal{F}$ of two SNARKs $\mathcal{I}$ and $\mathcal{O}$ (Section 18.1), we presented $\mathcal{F}$ in a manner that hopefully made intuitively clear that it is knowledge-sound: the $\mathcal{F}$-prover $\mathcal{P}_{\mathcal{F}}$ establishes using the outer SNARK $\mathcal{O}$ that it knows a proof π that would have caused the $\mathcal{I}$-verifier to accept the claim at hand, namely that $\mathcal{P}_{\mathcal{F}}$ knows a w such that $\mathcal{C}(w) = 1$. In turn, since $\mathcal{I}$ is knowledge-sound, any efficient party $\mathcal{P}_{\mathcal{F}}$ who knows such a proof π must also know such a witness w.[3]

Still, it is instructive to carefully write out a description of the procedure $\mathcal{E}_{\mathcal{F}}$ that extracts the witness w from $\mathcal{P}_{\mathcal{F}}$. This will help us understand knowledge-extraction for the "deeper" compositions considered in this section. As we will see, the natural knowledge extractor for a composed SNARK will have runtime that grows exponentially with the depth of the composition. This means that super-constant depth compositions will yield a superpolynomial-time knowledge-extractor. Hence, the knowledge-soundness of such deep compositions is not on firm theoretical footing.[4]

Knowledge extractor for $\mathcal{F} = \mathcal{O} \circ \mathcal{I}$. Given an efficient prover $\mathcal{P}_{\mathcal{F}}$ that can generate accepting proofs for $\mathcal{F}$, $\mathcal{E}_{\mathcal{F}}$ must identify a witness w such that $\mathcal{C}(w) = 1$. $\mathcal{E}_{\mathcal{F}}$ works as follows. Since a convincing proof for $\mathcal{F}$ establishes via the outer SNARK system $\mathcal{O}$ that $\mathcal{P}_{\mathcal{F}}$ knows a proof

[2] By in its own head, we mean the prover performs a computation without sending the result to the verifier.

[3] Note that if $\mathcal{O}$ satisfies only standard soundness rather than knowledge-soundness, the composed proof system $\mathcal{O} \circ \mathcal{I}$ may not even satisfy standard soundness. This is because $\mathcal{O}$ will only establish the *existence* of a proof π that would have caused the $\mathcal{I}$-verifier to accept. And there will typically *exist* convincing proofs of false statements under the SNARK $\mathcal{I}$: computational soundness of $\mathcal{I}$ only guarantees that such proofs are difficult for a cheating prover to find.

[4] While we cannot prove knowledge-soundness of superconstant-depth SNARK recursions, that does not necessarily mean we think deep recursions are *not* knowledge-sound, just that we don't know how to provably reduce their knowledge-soundness to that of the underlying base SNARK. Indeed deep recursions of SNARKs are beginning to see practical deployment in distributed environments (e.g., [65]. See also Sections 18.4 and 18.5).

π causing the $\mathcal{I}$-verifier to accept, $\mathcal{E}_{\mathcal{F}}$ can first apply the following sub-routine: "run the knowledge-extractor $\mathcal{E}_{\mathcal{O}}$ for $\mathcal{O}$ to extract from $\mathcal{P}_{\mathcal{F}}$ such a proof π". This sub-routine itself represents an efficient convincing prover algorithm $\mathcal{P}_{\mathcal{I}}$ for the inner SNARK $\mathcal{I}$. Hence, $\mathcal{E}_{\mathcal{F}}$ can apply the knowledge-extractor $\mathcal{E}_{\mathcal{I}}$ to extract from $\mathcal{P}_{\mathcal{I}}$ a witness w such that $\mathcal{C}(w) = 1$.

How efficient is $\mathcal{E}_{\mathcal{F}}$? $\mathcal{E}_{\mathcal{F}}$ has to apply the inner-SNARK knowledge-extractor $\mathcal{E}_{\mathcal{I}}$ to a prover $\mathcal{P}_{\mathcal{I}}$ that itself runs the outer-SNARK knowledge-extractor $\mathcal{E}_{\mathcal{O}}$ on $\mathcal{P}_{\mathcal{F}}$. Hence, $\mathcal{E}_{\mathcal{F}}$ may be significantly slower than $\mathcal{E}_{\mathcal{I}}$ or $\mathcal{E}_{\mathcal{O}}$ individually (though $\mathcal{E}_{\mathcal{F}}$ still runs in polynomial time as long as $\mathcal{E}_{\mathcal{I}}$ and $\mathcal{E}_{\mathcal{O}}$ both do). For example, if A denotes the number of times $\mathcal{E}_{\mathcal{I}}$ calls[5] $\mathcal{P}_{\mathcal{I}}$ to extract w from it, and B denotes the number of times $\mathcal{E}_{\mathcal{O}}$ calls $\mathcal{P}_{\mathcal{F}}$ to extract π from it, then the entire extraction procedure $\mathcal{E}_{\mathcal{F}}$ may call $\mathcal{P}_{\mathcal{F}}$ up to $A \cdot B$ times.[6]

Knowledge extractor for deeper compositions. Now consider a SNARK $\mathcal{O}$ composed with itself, say, four times, and denote the composition by $\mathcal{O}^4 := \mathcal{O} \circ \mathcal{O} \circ \mathcal{O} \circ \mathcal{O}$. We can view $\mathcal{O}^4$ as $\mathcal{O}^2 \circ \mathcal{O}^2$, where $\mathcal{O}^2 := \mathcal{O} \circ \mathcal{O}$. The previous paragraph shows that if A denotes the number of times that the knowledge extractor $\mathcal{E}_{\mathcal{O}}$ for $\mathcal{O}$ must run a convincing prover $\mathcal{P}_{\mathcal{O}}$ to extract a witness, then the number of times that the natural knowledge extractor for $\mathcal{O}^2$ must run a convincing prover $\mathcal{P}_{\mathcal{O}^2}$ to extract a witness is A^2. Then applying the same analysis to $\mathcal{O}^2 \circ \mathcal{O}^2$ means that the number of times the natural knowledge extractor for $\mathcal{O}^4$ must run a convincing prover is A^4.

In general, composing $\mathcal{O}$ with itself t times will yield a knowledge extractor that runs a prover generating convincing proofs at most A^t times. If A is polynomial in the size of the statement that the SNARK is applied to, then A^t will be superpolynomial unless t is constant.

[5] When we say that a knowledge extractor "calls" a prover more than once, we refer to the fact that the extractor might repeatedly "rewind and restart" the prover from which it is extracting a witness. We saw examples of this in the context of forking-lemma-based extractors for Σ-protocols (see Remark 12.1 in Section 12.2.1 and Section 14.4.1), and for SNARKs obtained thereof via the Fiat-Shamir transformation (Section 12.2.3).

[6] A and B may depend on the size of the statement being proven, but we suppress this dependence from our notation for simplicity.

Practical considerations of composition. For many popular SNARKs $\mathcal{O}$, there can be considerable concrete overhead in attempting to represent the $\mathcal{O}$-verifier as an equivalent instance of arithmetic circuit-satisfiability or R1CS, or whatever intermediate representation is "consumed" by the outer SNARK. Here, we highlight one particularly common and important issue, and describe how it is has been addressed to date.

As we have seen in Sections 14 and 17, many popular SNARKs require the verifier to perform operations in cryptographic groups in which the discrete logarithm problem is intractable (and for many SNARKs, the groups must furthermore be pairing-friendly, see Section 15.1). Modern instantiations of such cryptographic groups use elliptic curves (Section 12.1.2.2). Recall that elements of an elliptic curve group correspond to pairs of points $(x, y) \in \mathbb{F} \times \mathbb{F}$ that satisfy an equation of the form $y^2 = x^3 + ax + b$ for field elements a and b. $\mathbb{F}$ is referred to as the *base field* of the curve. When designing a discrete-logarithm-based SNARK for arithmetic circuit-satisfiability or R1CS-satisfiability over a field $\mathbb{F}_p$ of prime order p, one requires that the *order* of the elliptic curve group $\mathbb{G}$ be p (in this case, $\mathbb{F}_p$ is called the *scalar field* of $\mathbb{G}$). The crucial point here is that the base field $\mathbb{F}$ and the scalar field $\mathbb{F}_p$ of $\mathbb{G}$ are *not* the same field (see Section 12.1.2.2). This means that, in a discrete-log-based SNARK $\mathcal{O}$ for an arithmetic circuit $\mathcal{C}$ defined over field $\mathbb{F}_p$, the verifier has to perform field operations over a base field $\mathbb{F}$ that *differs* from $\mathbb{F}_p$.

Recall that in order to compose a SNARK $\mathcal{O}$ for circuit-satisfiability with itself, one must represent the verification procedure of $\mathcal{O}$ as an arithmetic circuit $\mathcal{C}'$ to which $\mathcal{O}$ can be applied. If $\mathcal{O}$ uses a cryptographic group $\mathbb{G}$ as per the above paragraph, then it is natural to define $\mathcal{C}'$ over the base field $\mathbb{F}$ of $\mathbb{G}$ rather than the scalar field $\mathbb{F}_p$ of $\mathbb{G}$, so that $\mathcal{C}'$ can "natively" perform the operations over $\mathbb{F}$ required to perform group operations in $\mathbb{G}$ (while it is possible to "implement" $\mathbb{F}$ operations via a circuit defined over a different field $\mathbb{F}_p$ using techniques discussed in Section 6, it is currently quite expensive, despite efforts from many researchers to make it less so). But in order to apply $\mathcal{O}$ to $\mathcal{C}'$, one needs to know *another* cryptographic group $\mathbb{G}'$ whose scalar field (rather than base field) is $\mathbb{F}$.

Accordingly, to support arbitrary-depth compositions of $\mathcal{O}$ with itself (or with other SNARKs), it is useful to identify a *cycle* of elliptic curves. The simplest form of such a cycle has length two. This is a pair of elliptic curve groups $\mathbb{G}$ and $\mathbb{G}'$ such that the base field $\mathbb{F}_p$ of $\mathbb{G}$ is the scalar field $\mathbb{F}$ of $\mathcal{G}'$ and vice versa. Using such a cycle of elliptic curves ensures that the verifier of $\mathcal{O}$ applied to a circuit over field $\mathbb{F}$ can be efficiently implemented via a circuit over field $\mathbb{F}_p$, and vice versa.

To walk through the specific example of depth-two recursive composition: let $\mathcal{O}$ be a SNARK for arithmetic circuit-satisfiability. It will be helpful to use a subscript $\mathcal{O}_{\mathbb{F}}$ to clarify what field the circuit-satisfiability instance is defined over. Then $\mathcal{O}^3 := \mathcal{O}_{\mathbb{F}_p} \circ \mathcal{O}_{\mathbb{F}} \circ \mathcal{O}_{\mathbb{F}_p}$ will work as follows to establish knowledge of a w such that $\mathcal{C}(w) = 1$, where $\mathcal{C}$ is defined over $\mathbb{F}_p$. First, the $\mathcal{O}^3$ prover $\mathcal{P}$ in its own head will generate a proof π that convinces the $\mathcal{O}_{\mathbb{F}_p}$-verifier of the claim. The $\mathcal{O}_{\mathbb{F}_p}$ verifier for this claim can be efficiently represented by a circuit $\mathcal{C}'$ over $\mathbb{F}$. So (in its own head once again) the $\mathcal{O}^3$ prover will generate an $\mathcal{O}_{\mathbb{F}}$-proof π' that it knows such an $\mathcal{O}_{\mathbb{F}_p}$-proof π. The $\mathcal{O}_{\mathbb{F}}$-verifier for this claim can in turn be efficiently implemented by a circuit $\mathcal{C}''$ over $\mathbb{F}_p$, so the $\mathcal{O}^3$ prover finally computes a proof π'' that it knows such an $\mathcal{O}_{\mathbb{F}}$-proof π'. And $\mathcal{P}$ sends this proof explicitly to the $\mathcal{O}^3$ verifier.

More generally, given a cycle of elliptic curves, arbitrary-depth composition of $\mathcal{O}_{\mathbb{F}}$ and $\mathcal{O}_{\mathbb{F}_p}$ can be supported. Every time the prover needs to produce a proof π' that it knows a proof π that the $\mathcal{O}_{\mathbb{F}_p}$-verifier would accept, it represents the $\mathcal{O}_{\mathbb{F}_p}$-verifier as a circuit over $\mathbb{F}$ and applies the $\mathcal{O}_{\mathbb{F}}$ SNARK to this circuit, and similarly with the roles of $\mathbb{F}_p$ and $\mathbb{F}$ reversed.

Currently, a popular cycle of (non-pairing-friendly) curves are Pasta curves,[7] which are reasonably close in efficiency to some of the best curves that don't support cycles (e.g., Curve25519, see Section 12.1.2.2). Cycles of pairing-friendly curves are also known, e.g., via so-called MNT curves [96], but, at the time of writing, for a given security level these remain significantly less efficient than popular pairing-friendly curves for SNARK design that don't support cycles (e.g., BLS12-381, see Section 15.1). This owes to a need of the cycle-supporting curves

[7] https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/.

to work over significantly larger finite fields, which leads to slower group operations. While *cycles* of pairing-friendly curves are currently very expensive, efficient depth-one composition of two pairing-based SNARKs does not require a cycle of curves; rather, it only requires two pairing-friendly curves such that the base field of one is the scalar field of the other. This is currently offered by an efficient curve known as BLS12-377 and a sister curve called BW6-761 [74], [154].

Another common practical consideration arising in recursive SNARK composition is that the verifier in many transparent SNARKs performs Merkle hash path verifications, which means cryptographic hash operations must be expressed as a circuit- or R1CS-satisfiability instance. As mentioned in Section 6, there has been considerable effort devoted to developing "SNARK-friendly" hash functions, meaning plausibly collision-resistant hash functions that can be efficiently expressed in such a form.

18.3 Other Applications of SNARK Composition

We have seen that composition of SNARKs can be used to improve efficiency: a SNARK with fast prover and somewhat slow verification can be composed with itself or with another SNARK to improve the verification costs. There are other reasons to compose SNARKs.

Incremental computation. One, which we detail later in this section (Sections 18.4 and 18.5), uses recursion more directly to construct efficient SNARKs tailored for iterative computation, i.e., to prove that for some designated input x and specified function F that $F(F(F(F(F(F(x))))))) = y$. More generally, let $F^{(i)}(x)$ denote the i-fold iterative application of F to x, e.g., $F^{(3)}(x) = F(F(F(x)))$. A quintessential application of such proof systems is to let F be a *delay function*, meaning a simple function that requires some non-trivial sequential computation to compute. Then a SNARK for many iterative applications of F yields a *verifiable delay function*: a function that requires substantial sequential time to compute, the result of which can be verified very quickly.

Incrementally Verifiable Computation (IVC). Certain applications (to be discussed momentarily) actually call for a primitive called *incrementally verifiable computation* [239]. This means that after each application j of F to x, a prover can output y_j and a SNARK proof π_j that $F^{(j)}(x) = y_j$, and moreover, given y_j and π_j, *any* other party can apply F to y_j to obtain an output y_{j+1} and efficiently compute a new SNARK proof π_{j+1} that $F^{(j+1)}(x) = y_{j+1}$.

Applications to distributed computing environments. In fact, our SNARKs for iterative computation will be able more generally to handle non-deterministic computations F. That is, F can take two inputs, a public input x and a witness w, and produce some output $y = F(x, w)$. The SNARKs we present hereon in this section will be able to[8] establish knowledge of witnesses $w_1, \ldots, w_i$ such that

$$F(F(\ldots F(F(F(x, w_1), w_2), w_3), \ldots, w_{i-1}), w_i) = y_i.$$

Here is one example of a possible application to public blockchains. Think of F as taking as input the current state of an "accumulation" (e.g., Merkle-hash, see Section 7.3.2.2) of all account balances for a public blockchain, and think of each witness w_i as specifying a new valid transaction t_i along with associated proof-of-work, and such that F outputs an updated accumulation (i.e., F outputs the accumulation of the new account balances following the processing of transaction t_i). Then a SNARK for the above yields a proof that y_i is a valid accumulation of account balances after i transactions. This can enable computationally weak nodes in a blockchain network to very efficiently learn from any untrusted party an accumulation of the global state of the network (i.e., the current account balances), with a proof that the accumulation actually captures a sequence consisting of a certain number of valid transactions and associated proofs-of-work. This may be important for protocols that designate the current state of the network to be that of the "longest chain", i.e., the longest known sequence of valid transactions. Hence, nodes can trustlessly learn the accumulation

[8] For simplicity, we do not present the SNARKs in this level of generality but they will support it without modification.

of the network state, with no need to download the entire transaction history of the network or even the current account balances.

Proof aggregation. Another application of SNARK composition is proof aggregation, which can be explained via the following example application. Suppose that a prover $\mathcal{P}$ claims for some public input x and function F that $F(x) = y$, but computing F is highly computation-intensive. Imagine that the computation is broken up into ℓ more manageable pieces, say, $F_1(x), \ldots, F_\ell(x)$, that can be performed independently of each other. $\mathcal{P}$ farms each piece out to a different machine (possibly untrusted even by the prover, who is in turn untrusted by the verifier), to produce outputs $y_1, \ldots, y_\ell$, which are then combined via some aggregation function G to produce the final output y.

In order to prove that $F(x) = y$, each machine can produce a proof π_i that $y_i = F_i(x)$, and send both π_i and the result y_i back to $\mathcal{P}$. Then it suffices for $\mathcal{P}$ to (a) prove knowledge of the convincing proofs $\pi_1, \ldots, \pi_\ell$ for the ℓ claims $y_i = F_i(x)$, and (b) prove that $G(y_1, \ldots, y_\ell) = y$. One can accomplish this by applying a SNARK to the computation that first *verifies* the proofs $\pi_1, \ldots, \pi_\ell$ and then computes $G(y_1, \ldots, y_\ell)$.

18.4 SNARKs for Iterative Computation via Recursion

Recall that $F^{(i)}(x)$ denote the i-fold iterative application of F to x. Suppose we want to design a SNARK for the claim that $F^{(i)}(x) = y$.

One could of course apply any of the (non-composed) SNARKs from earlier sections of this survey to $F^{(i)}$, but these come with various downsides and tradeoffs, delineated in detail in the next section, Section 19. For starters, they do not support IVC (Section 18.3). Turning to efficiency, if one desires the shortest possible proofs and fastest verification, the SNARKs with these properties require a trusted setup (see Section 17). They also tend to be quite space-intensive for the prover due in part to their use of FFTs, so applying them to very large computations may not be feasible, and their use of pairing-friendly groups can lead to slow prover time. While many of the transparent SNARKs of earlier sections avoid FFTs and pairings, they have much

larger proofs and verification costs than the trusted-setup SNARKs with fastest verification.

The recursive-composition-of-SNARKs approach. Can we address the above issues by taking a base SNARK $\mathcal{O}$ and applying recursive composition? Let us imagine for a moment that we have already designed a SNARK $\mathcal{O}_{i-1}$ for the claim that $F^{(i-1)}(x) = y_{i-1}$. Then here is a SNARK $\mathcal{O}_i$ for the claim that $F^{(i)}(x) = y_i$: the prover $\mathcal{P}$ uses the base SNARK $\mathcal{O}$ to prove that

(a) it knows an $\mathcal{O}_{i-1}$-proof π_{i-1} that $F^{(i-1)}(x) = y_{i-1}$, and

(b) that $F(y_{i-1}) = y_i$.[9]

This recursive-composition-of-SNARKs approach to incremental computation has been pursued (e.g., [45]) using the trusted-setup SNARK with fastest known verification, which is now due to Groth [142] (Section 17.5.6).[10] A major benefit of the recursive approach is that it yields

[9] An important practical issue here is that, in order to identify a single arithmetic circuit confirming both (a) and (b), it is essential that the $\mathcal{O}_{i-1}$-verifier's computation and F itself both be efficiently expressible as a circuit over the same field. This can be challenging for SNARKs that perform elliptic curve operations, because as discussed in Section 18.2, such SNARK verifiers are only efficiently representable as circuits over the *base* field $\mathbb{F}$ of the curve, which differs from the (scalar) field $\mathbb{F}_p$ that F is presumably efficiently representable over. One way to sidestep this issue is to identify a cycle of curves with scalar and base fields $\mathbb{F}_p$ and $\mathbb{F}$ such that F is efficiently computable by circuits over *both* $\mathbb{F}_p$ and $\mathbb{F}$. This way, at each step i, the $\mathcal{O}_{i-1}$ verifier will be efficiently expressible as a circuit over one of the two fields (which one depends on whether i is odd or even), and F will also be efficiently expressible as a circuit over the same field. If F is only efficiently computable by a circuit over $\mathbb{F}_p$, then one will run into the issue that the $\mathcal{O}_{i-1}$-verifier is efficiently representable as a circuit only over $\mathbb{F}$, and F itself is not. To address this, one can define $\mathcal{O}_i$ via *two* steps of SNARK composition, rather than one. In the first step, the prover represents the $\mathcal{O}_{i-1}$-verifier as a circuit over $\mathbb{F}$, and in its own head computes an $\mathcal{O}_{\mathbb{F}}$-proof π that it knows an $\mathcal{O}_{i-1}$-proof π_{i-1} that $F^{(i-1)} = y_{i-1}$. Then, since (unlike the $\mathcal{O}_{i-1}$-verifier) the $\mathcal{O}_{\mathbb{F}}$-verifier *is* efficiently representable as a circuit over $\mathbb{F}_p$, there is a small circuit over $\mathbb{F}_p$ to establish that both (a) the prover knows such a proof π and and (b) that $F(y_{i-1}) = y_i$. Hence, $\mathcal{O}_{\mathbb{F}_p}$ can be applied to this circuit to yield a proof that $F^{(i)}(x) = y_i$.

[10] More recent work has studied recursive-composed SNARKs with a universal rather than circuit-specific trusted setup, but this leads to even higher overheads for the prover [96].

IVC (Section 18.3): for each iteration $j-1$, the prover could output y_{j-1} and the proof π_{j-1} that $y_{j-1} = F^{(j-1)}(x)$, and any other party could "pick up the computation from there", computing $F(y_{j-1})$ and using π_{j-1} to compute a proof π_j that $y_j = F^{(j)}(x)$.

Relative to the direct application of the non-composed base SNARK, the above recursive solution also reduces the prover's space cost, because the prover only ever applies the base SNARK to a *single* application of F, one after the other (i.e, it does not apply the base SNARK "all at once" to an entire circuit computing $F^{(i)}$). That is, at any time j during its computation of the proof π_i, the prover only needs to remember the preceding proof π_{j-1} and the preceding output y_{j-1} of $F^{(j-1)}$.

On the other hand, a significant downside of the recursive approach when applied to a SNARK that uses pairings such as Groth's [142] is that the prover is quite slow, in large part owing to the need to use cycles of pairing-friendly elliptic curves to support arbitrary-depth recursion (Section 18.2). On top of this, there is additional overhead for the prover that can be traced to a notion we term the *overhead of recursion*.

The overhead of recursion. Effectively, the final SNARK proof π_i for $F^{(i)}$ establishes that for all $j \leq i$, the prover $\mathcal{P}$ not only faithfully applied F to y_{j-1} to obtain y_j (as per (b) above), but *also* that $\mathcal{P}$, in its own head, faithfully verified the proof π_{j-1} as per (a) above.[11] Put another way, the above recursive approach replaces the computation of $F(y_{j-1})$ with a *larger* computation $F'(y_{j-1}, \pi_{j-1})$ that outputs $F(y_{j-1})$ and verifies π_{j-1}, and it applies the base SNARK to F' for all $j \leq i$. (This perspective will come up again in Section 18.5).

We refer to the added cost to the prover of establishing that it verified π_{j-1} for each iterative application j of F as the "overhead of recursion". This is because non-recursive solutions—i.e., a direct application of a SNARK to a circuit computing $F^{(i)}$—require the prover to establish only that it faithfully applied F all i times, not that it verified any proofs of its own faithfulness along the way. Hence, the

[11] To clarify, π_i establishes all of this without even "telling the verifier" what y_{j-1} or π_{j-1} even were.

"overhead of recursion" is purely extra work for the prover, which does not arise in non-recursive solutions.

This overhead is naturally measured by the number of gates in a circuit, or other intermediate representation as appropriate, implementing the base SNARK's verifier.[12] This will be the dominant contributor to the prover's costs if this circuit is larger than the circuit required to implement F itself. Specifically, this happens if the circuit representing F is smaller than the *recursion threshold* of the base SNARK $\mathcal{O}$ (see Section 18.2).

Trusted-setup SNARKs with state-of-the-art verification costs [142] have a reasonably low recursion threshold. Still, we will see later (Section 18.5) that this overhead can be reduced further via other approaches that moreover can avoid a trusted setup and pairing-friendly groups (the use of pairings both increases the recursion threshold and, as mentioned above, leads to concretely high prover costs).[13]

Recursively composing transparent SNARKs. To recap, there are a number of downsides to above approach of recursively composing a SNARK with state-of-the-art verification costs: the base SNARK's need for a trusted setup, the very high prover overheads due to the use of cycles of pairing-friendly curves, and the concretely sub-optimal "overhead of recursion".

The most straightforward approach to address the first two issues is to replace the trusted-setup SNARKs with transparent SNARKs that moreover do not require pairing-friendly groups. These SNARKs all utilize transparent polynomial commitment schemes—e.g., based on FRI (Section 10.4), Ligero's polynomial commitment scheme (Section 10.5),

[12] The issue described at the end of Footnote 9 can further increase the overhead of recursion, by forcing *two* statements about SNARK verification circuits to be proved for every application of F, rather than one.

[13] Verification of Groth's SNARK [142] involves 3 pairing computations, which are concretely fairly expensive, especially once represented as a circuit or R1CS. Hence, there is room to reduce this overhead further. We will see an approach later in this section (Section 18.5) that reduces the "three pairing computations" down to roughly two group exponentiations in a *non*-pairing-friendly group, which concretely can be represented by a significantly smaller circuit or R1CS than three pairing computations.

Hyrax's polynomial commitment scheme (Section 14.3), or Bulletproofs (Section 14.4). The problem with a naive implementation of this approach is that the verification of evaluation proofs of such polynomial commitment schemes is quite expensive and hence the overhead of recursion is very large. For example, if the popular Bulletproofs polynomial commitment is used, then while proofs are short (logarithmic in size), the verification cost is linear. Even FRI-based polynomial commitments (Section 10.4.4), while achieving polylogarithmic verification time, has proofs that are concretely quite large for appropriate security levels, and verification involve many Merkle hash path authentication operations, which can be somewhat expensive to represent as a circuit or R1CS (see the end of Section 18.2).

To address the overhead of recursion in this case, a line of works starting with Halo [64], [75], [81], [82], [175] has roughly shown how to avoid feeding verification of evaluation proofs of polynomial commitment schemes through the proof machinery. The verifier in these transparent SNARKs can be split into two parts: (a1) verifying all parts of the proof other than evaluations of committed polynomials and (a2) verifying evaluations of committed polynomials. Essentially, the SNARK is modified to simply *omit* the verification check (a2). This means that, each time the prover, in its own head, generates a "proof"[14] π_j that $F^{(j)}(x) = y_j$ (having already computed a "proof" π_{j-1} that $F^{(j-1)}(x) = y_{j-1}$), π_j does *not* directly attest to the validity of any claimed evaluations of committed polynomials involved in the "proof". So these evaluation claims must be checked separately. What these works roughly do is show how to use homomorphism properties of known polynomial commitment schemes to cheaply "batch-check" *all* evaluations of *all* committed polynomials across all "proofs" $\pi_1, \ldots, \pi_i$ that the prover generated in its own head. That is, all such evaluation claims regarding committed polynomials across $\pi_1, \ldots, \pi_i$ are "accumulated" into a single claim, which can then be checked at the same cost as a *single* claim. In Section 16.1, we covered details of this technique in

[14] Here, we are putting the word "proof" in quotes, because π_j omits essential verification information, namely verification of evaluations of committed polynomials. Hence, π_j is not actually a complete SNARK proof for the claim at hand, that $F^{(j)}(x) = y_j$.

the case of homomorphic polynomial commitments all being evaluated at the same point.

The most recent works in this line have taken the above approach to its logical extreme and derived SNARKs for iterative computation $F^{(i)}$ purely from homomorphic vector commitment schemes (i.e., without first developing a "base SNARK" that is recursively applied i times). See Footnote 24 in Section 18.5.4 for additional discussion of this perspective. The following section describes one such result, yielding a proof system called Nova [175].

18.5 SNARKs for Iterative Computation via Homomorphic Commitments

Our goal in this section is to design a SNARK for iterative computation directly from homomorphic vector commitment schemes. The resulting SNARK is transparent, avoids the need for pairing friendly curves, and has state-of-the-art overhead of recursion. These last two properties together ensure a significantly faster prover relative to the recursive composition of pairing-based SNARKs (Section 18.4).

18.5.1 Informal Overview of the SNARK

The SNARK will roughly work as follows. Using the front-end techniques of Section 6, one first transforms F into an equivalent R1CS instance, i.e., three public matrices $A, B, C \in \mathbb{F}^{n \times n}$ such that $F(x) = y$ if and only if there exists a vector z of the form (x, y, w) for some witness w such that $(A \cdot z) \circ (B \cdot z) = C \cdot z$. Here $\circ$ denotes the element-wise product of two vectors.[15]

[15] Previous sections in this section referred to SNARKs for arithmetic circuit satisfiability for simplicity and concreteness, but as pointed out in Footnote 1, they apply without modification to SNARKs for R1CS. In this section, we use the formalism of R1CS rather than circuits because Nova is most naturally described in the R1CS setting. Of course, R1CS is a generalization of a circuit (see Section 8.4), so any SNARK for R1CS representations also yields a SNARK for circuit representations.

Let $y_0 = x$. Then proving that $F^{(i)}(x) = y_i$ is equivalent to showing the existence of vectors $w_1, \ldots, w_i$ such that for

$$z_j \coloneqq (y_{j-1}, y_j, w_j), \tag{18.1}$$

$$(A \cdot z_j) \circ (B \cdot z_j) = C \cdot z_j \colon j = 1, \ldots, i. \tag{18.2}$$

The rough idea of the SNARK is that $\mathcal{P}$ will commit to all of the vectors $z_1, \ldots, z_i$ using a homomorphic vector-commitment scheme, and prove that each one has the form Equation (18.1) and satisfies Equation (18.2). It will do this by repeatedly applying a primitive called a "folding scheme"—roughly, a way of taking two R1CS instances of the form Equation (18.2) and transforming them into a single R1CS instance such that the derived instance is satisfied if and only if both original instances are satisfied.[16] The folding scheme can be repeatedly applied to reduce all i instances of Equation (18.2) into a single instance. For simplicity, we will focus on the "sequential" folding pattern whereby instance one of Equation (18.2) is folded with instance two, and then the resulting derived instance is folded with instance three, and then the resulting derived instance is folded with instance four, and so on until all i instances have been folded into a single one.[17] The folding scheme is interactive, but the interaction can be removed with the Fiat-Shamir transformation.

The validity of this final R1CS instance can be proven with any SNARK for R1CS instances of the form $(A \cdot z) \circ (B \cdot z) = C \cdot z$ in which the prover commits to the witness vector z via the same homomorphic vector commitment scheme used by the prover to commit to $z_1, \ldots, z_i$. This includes, for example, SNARKs that make use of the Bulletproofs

[16] This folding scheme is reminiscent several earlier protocols in this text. Most directly, in each round of Bulletproofs (Section 14.4), a claim about an inner product of committed vectors of length n is reduced to a derived claim about an inner product of vectors of length $n/2$. Also, in each round of the sum-check protocol (Section 4.2), a claim about a sum over 2^ℓ terms is reduced to a claim about a sum over $2^{\ell-1}$ terms. In fact, there have been works that view these protocols through a unified lens [70], [174].

[17] In general, any folding pattern can be used. That is, we can treat the i instances as the leaves of any binary tree, with any internal node of the tree representing the "folding" of its two children into a single instance. The root of the tree represents the final R1CS instance that results from all of the folding operations.

polynomial commitment scheme (Section 14.4) as the commitment in Bulletproofs is just a generalized Pedersen commitment to the coefficient vector of the polynomial. If Bulletproofs is used, the length of the SNARK proof for the final R1CS instance that results from folding can be made $O(\log n)$, though the verification time will be $O(n)$.[18]

The above brief description glosses over a number of details. First, the folding scheme will take two R1CS instances and not yield another R1CS instance, but rather a generalization that we call *committed-R1CS-with-a-slack-vector*. Second, because each folding operation will require a message from the prover to the verifier (and a random challenge sent from verifier to prover), the proof length of the resulting protocol will be linear in i, when we would really like a proof length that is *independent* of i. We will ultimately achieve the desired proof length via a variant of recursive proof composition (Section 18.5.4). We additionally have not explained how to check that each committed vector z_j has the form of Equation (18.1).

18.5.2 A Folding Scheme for Committed-R1CS-with-a-Slack-Vector

The problem of committed-R1CS-with-a-slack-vector. In an instance of this problem, there are three public $n \times n$ matrices A, B, and C with entries from a field $\mathbb{F}$, as well as a public scalar $u \in \mathbb{F}$ and a public vector $s \in \mathbb{F}^m$. In addition to those public objects, there are two committed vectors $w \in \mathbb{F}^{n-m}$ and E in $\mathbb{F}^n$. Let $z = (s, w) \in \mathbb{F}^n$. One should think of the prover as having already committed to w and E using a homomorphic vector-commitment scheme (e.g., Pedersen vector commitments from Section 14.2). The prover claims that

$$(A \cdot z) \circ (B \cdot z) = u \cdot (C \cdot z) + E.$$

Folding two instances. Consider having two instances of committed-R1CS-with-a-slack-vector, in which the public matrices in the two

[18] For iterative computation, one typically thinks of the number of iterations i as very large, and function F applied at each iteration as small, perhaps even computed by a constant-size circuit. In this case, $O(n)$ can be thought of as a constant and $O(\log n)$ as an *even smaller* constant.

instances are identical. That is, the prover has claimed that:

$$(A \cdot z_1) \circ (B \cdot z_1) = u_1 \cdot C \cdot z_1 + E_1, \tag{18.3}$$

$$(A \cdot z_2) \circ (B \cdot z_2) = u_2 \cdot C \cdot z_2 + E_2. \tag{18.4}$$

Here, $A, B, C \in \mathbb{F}^{n \times n}$ are public matrices, $u_1, u_2 \in \mathbb{F}$ are public scalars, $s_1, s_2 \in \mathbb{F}^m$ are public vectors, $w_1, w_2 \in \mathbb{F}^{n-m}$ and $E_1, E_2 \in \mathbb{F}^n$ are committed vectors, and $z_1 = (s_1, w_1)$ and $z_2 = (s_2, w_2)$. $\mathcal{V}$ would like to check both of these claims. The naive way to do this would be to have the prover open the commitments to w_1, w_2, E_1, and E_2, so $\mathcal{V}$ can check both claims directly, but this naive approach is too expensive for our purposes. Instead, imagine the verifier $\mathcal{V}$ would like to "take a random linear combination" of the two claims, to derive a single claim of the same form, such that the derived claim is true (up to some negligible soundness error) if and only if both of the original claims are true.

Here is a way the verifier could try to accomplish this.

A first attempt that doesn't work. The verifier could choose a random field element $r \in \mathbb{F}$, and let

$$s \leftarrow s_1 + r \cdot s_2 \tag{18.5}$$

$$w \leftarrow w_1 + r \cdot w_2 \tag{18.6}$$

$$u \leftarrow u_1 + r \cdot u_2 \tag{18.7}$$

$$E \leftarrow E_1 + r^2 E_2. \tag{18.8}$$

Observe that $\mathcal{V}$ can directly compute s and u because $s_1, s_2 \in \mathbb{F}^m$ and $u_1, u_2 \in \mathbb{F}$ are public. Also, by homomorphism of the commitment scheme used by $\mathcal{P}$ to commit to w_1, w_2, E_1, and E_2, the verifier can on its own compute commitments to w and E. The verifier might *hope* that under these definitions, Equation (18.3) and (18.4) imply the following (and vice versa):

$$(A \cdot z) \circ (B \cdot z) = u \cdot (C \cdot z) + E. \tag{18.9}$$

If this were the case, then the verifier, on its own, could derive a single new instance of committed-R1CS-with-a-slack-vector that is *equivalent* to the validity of the two original instances (Equations (18.3) and (18.4)).

Unfortunately, even if Equation (18.3) and (18.4) both hold, Equation (18.9) does *not* hold. But as we will see, we can slightly modify the definition of E so that Equation (18.9) does hold.

What does work. Let us redefine E to include an extra "cross-term", namely, throw away Equation (18.8) and replace it with:

$$E \leftarrow E_1 + r^2 E_2 + r \cdot T \tag{18.10}$$

where

$$T \leftarrow (A \cdot z_2) \circ (B \cdot z_1) + (A \cdot z_1) \circ (B \cdot z_2) - u_1 \cdot C \cdot z_2 - u_2 \cdot C \cdot z_1. \tag{18.11}$$

Then it can be checked via elementary algebra that Equation (18.9) holds for every choice of $r \in \mathbb{F}$.

Calculation showing that Equation (18.9) holds for every $r \in \mathbb{F}$. The left hand side of Equation (18.9) is:

$$\begin{aligned}(A \cdot z) \circ (B \cdot z) &= (A \cdot z_1 + r \cdot A \cdot z_2) \circ (B \cdot z_1 + r \cdot B \cdot z_2) \\ &= (A \cdot z_1) \circ (B \cdot z_1) + r^2 \cdot (A \cdot z_2) \circ (B \cdot z_2) + r \\ &\quad \cdot ((A \cdot z_1) \circ (B \cdot z_2) + (A \cdot z_2) \circ (B \cdot z_1))\end{aligned} \tag{18.12}$$

while the right hand side equals:

$$\begin{aligned}u \cdot (C \cdot z) &= (u_1 + r u_2) \cdot C \cdot (z_1 + r z_2) \\ &= u_1 \cdot C \cdot z_1 + E_1 + r^2 (u_2 \cdot C \cdot z_2 + E_2) \\ &\quad + r \left(u_2 \cdot C \cdot z_1 + u_1 \cdot C \cdot z_2\right)\end{aligned} \tag{18.13}$$

By Equations (18.3) and (18.4), we can rewrite Expression (18.12) as:

$$\begin{aligned}&u_1 \cdot C \cdot z_1 + E_1 + r^2 \cdot (u_2 \cdot C \cdot z_2 + E_2) \\ &\quad + r \cdot ((A \cdot z_1) \circ (B \cdot z_2) + (A \cdot z_2) \circ (B \cdot z_1)) .\end{aligned} \tag{18.14}$$

The difference between Expression (18.14) and the right hand side of Equation (18.13) is exactly r times the value assigned to T by Equation (18.11).

Accordingly, consider the following simple interactive protocol that seeks to "reduce" checking that Equation (18.3) and (18.4) both hold to the task of checking that Equation (18.9) holds: First, $\mathcal{P}$ commits to a vector v claimed to equal the cross-term T (Equation (18.11)) using the same homomorphic vector-commitment scheme used to commit to w_1, w_2, E_1, and E_2. Next, $\mathcal{V}$ chooses r at random from $\mathbb{F}$ and sends it to $\mathcal{P}$. Observe that, given the commitments to E_1, E_2, and v, $\mathcal{V}$ can use the homomorphism to compute a commitment to the vector $E_1 + r^2 E_2 + r \cdot v$, which, if v is as claimed, equals the right hand side of Equation (18.10).

We have already explained that if the committed vector v equals T (Equation (18.11)) as prescribed, then Equation (18.10) holds with probability 1 over the random choice of r. Meanwhile, it is not hard to see that if the prover commits to a vector v that *differs* from T, then with probability $1 - 2/|\mathbb{F}|$ over the random choice of r, Equation (18.9) will fail to hold. This is because, if $v_j \neq T_j$ for some $j \in \{1, \ldots, n\}$, then the jth entries of the vectors on the left hand side and right hand side of Equation (18.9) will be two distinct degree-2 univariate polynomials in r, and hence will disagree at a randomly chosen input with probability $1 - 2/|\mathbb{F}|$. Here, it is essential that the prover is forced to commit to the cross-term vector T before learning the verifier's choice of $r \in \mathbb{F}$. Similarly, if either Equation (18.3) or (18.4) does not hold, then there is no vector T that the prover can commit to that would render every entry of the right hand and left hand sides of Equation (18.9) to be the same polynomials in r.

Formally, to be useful in designing a SNARK for iterative computation, we need to show that the above folding scheme is a proof of knowledge, meaning given any efficient prover that can convince the verifier of the validity of the folded instance with non-negligible probability, we can extract openings of the vectors w_1, E_1, w_2, E_2 that respectively satisfy the instances that were folded together (Equations (18.3) and (18.4)). We omit the details, as the paragraph above conveys the key intuition as to why a prover that does not behave as prescribed will, with overwhelming probability over the choice of r, be left to establish a

false claim after the folding, namely that it can open the commitments to w and E to vectors satisfying Equation (18.9).[19]

While this folding scheme is interactive, it is public coin, and hence can be rendered non-interactive via the Fiat-Shamir transformation (i.e, replace the verifier's challenge with a hash of the public inputs and the prover's message in the folding scheme).

18.5.3 A Large Non-Interactive Argument

A non-interactive argument of knowledge for an iterative computation $F^{(i)}(x)$ with proof length linear in i can be obtained by repeatedly applying the above folding scheme in the manner sketched in Section 18.5.1. This proof length is far too large to be interesting in applications, but it will be a useful object to have considered when we turn to designing the final SNARK (Section 18.5.4).

We will describe the proof as being produced and processed in "rounds", even though it is non-interactive. Since there is no message sent from $\mathcal{V}$ to $\mathcal{P}$, the entire proof is obtained by simply concatenating all prover messages across all "rounds".

At the start of each "round" $j > 1$ of the protocol, there is already a "running folded instance" I of committed-R1CS-with-slack-vector that captures the result of having folded across the first j rounds the R1CS instances capturing the first $j - 1$ applications of F (as per Equation (18.2)), and the purpose of round $j > 1$ is to fold into this running instance the R1CS capturing the jth application of F (again, as per Equation (18.2)). This means that at the start of round $j > 1$, the verifier will be tracking a commitment c_w to the "witness vector" w for I, and a commitment c_E for the "slack vector" E for I. The verifier at all times keeps track of the following variables:

- *round-count* (meant to track the number j of applications of F that have been processed so far).
- *prev-output* (meant to track $y_{j-1} = F^{(j-1)}(x)$)

[19] Readers are referred to the knowledge-soundness analysis of the Bulletproofs polynomial commitment (Section 14.4) for an example of a knowledge-soundness analysis for a folding scheme.

- *cur-output* (meant to track $y_j = F^{(j)}(x)$)
- $u \in \mathbb{F}$ (meant to track the scalar u of the running folded instance I)
- $s \in \mathbb{F}^m$ (meant to track the public input s to the running folded instance I)
- c_w (meant to track the commitment to the witness vector w of the running folded instance I)
- c_E (meant to track the commitment to the slack vector E of the running folded instance I).

Let us introduce some notation to capture this state of affairs at the start of round j. We denote the prover's claim in running folded instance I at the very start of round j by

$$(A \cdot z) \cdot (B \cdot z) = u \cdot (C \cdot z) + E, \tag{18.15}$$

with the verifier's variables c_w, c_E being a commitments to w and E respectively, and recall that $z = (s, w)$. As per Equation (18.2), there is an R1CS instance that is satisfiable if and only if $F(y_{j-1}) = y_j$. This R1CS instance has the form

$$(A \cdot z_j) \cdot (B \cdot z_j) = C \cdot z_j, \tag{18.16}$$

where $z_j = (s_j, w_j) \in \mathbb{F}^m \times \mathbb{F}^{n-m}$, and $s_j = (y_{j-1}, y_j)$. Let us refer to this R1CS instance as I_j.

The prover's work in round j. At the start of round j, the prover sends the claimed value of y_j. This reveals to the verifier the public vector $s_j = (y_{j-1}, y_j)$, as $\mathcal{V}$ learned the claimed value of y_{j-1} in the previous round. The prover also sends a commitment c_{w_j} to vector w_j. Together, these quantities specify the committed-R1CS instance I_j given in Equation (18.16). The purpose of round $j > 1$ is then to fold I_j into the running folded instance I. Accordingly, the prover sends a commitment c_T to the claimed cross-term T (Equation (18.11)). In round $j = 1$, there is no folding operation to perform, as the verifier will simply set the running folded instance to I_1; see next paragraph for details.

How the verifier $\mathcal{V}$ processes round j. Upon reading the prover's message in round $j = 1$, $\mathcal{V}$ sets its variables in accordance with the running folded instance becoming I_1. Specifically, $\mathcal{V}$ sets *round-count* to 1, *prev-output* to x, *cur-output* to the claimed value of y_1, u to 1, s to (*prev-output*, *cur-output*), c_w to c_{w_1}, and c_E to a commitment to the all-0s vector.

Upon receiving the prover's message in round $j > 1$, the verifier increments *round-count* from $j-1$ to j, sets *prev-output* to *cur-output*, and updates *cur-output* to (the claimed value of) y_j. In a truly interactive protocol, the verifier would randomly choose the field element $r \in \mathbb{F}$ used for that round's folding operation and send it to the prover, but in the non-interactive setting, both prover and verifier can determine r via the Fiat-Shamir transformation as per Section 18.5.2. After r is chosen, using homomorphism of the vector commitment scheme, $\mathcal{V}$ updates c_w to a commitment to $w + rw_j$ (as per Equation (18.6)).[20] $\mathcal{V}$ also updates c_E to a commitment to $E + rT$ (as per Equation (18.10)).[21] $\mathcal{V}$ updates $u \leftarrow u + r$ (as per Equation (18.7))[22] and updates $s \leftarrow s + r \cdot s_j$, where $s_j = (\textit{prev-output}, \textit{cur-output})$ (as per Equation (18.5)).

In this manner, after processing all i "rounds" of the proof, the verifier has computed a single folded committed-R1CS-with-slack-vector instance as per Equation (18.15), whose validity, up to a negligible soundness error, is equivalent to the validity of all i applications of F. In this final "round", the prover can establish the validity of the instance using any SNARK for committed-R1CS-with-slack-vector. Such a SNARK can in turn be easily obtained from any SNARK for R1CS satisfiability that commits to witness vectors via the same homomorphic vector commitment scheme used throughout the folding protocol. This includes the SNARK for R1CS from Section 8.4 when combined with, say, the Bulletproofs polynomial commitment scheme (Section 14.4).

[20] i.e., $c_w \leftarrow c_w \cdot (c_{w_j})^r$ where $\cdot$ denotes the group operation of the multiplicative group over which the Pedersen vector commitments used by the protocol are defined (see Section 14.2).

[21] Note that Equation (18.10) simplifies due to the fact that there is no slack vector in the R1CS instance of Equation (18.16)—equivalently, the slack vector is zero.

[22] Note that Equation (18.7) simplifies due to the fact that in the right hand side of Equation (18.16), $C \cdot z_j$ is multiplied by the trivial scalar 1.

18.5.4 The Final SNARK: Nova

Unfortunately, the argument of the previous section yields a proof π that grows linearly with i, the number of applications of F. Roughly speaking, we now address this by forcing the SNARK prover to, in its own head, perform the verifier's processing of π across i "rounds" of the protocol, and thereby avoid having the prover explicitly send π to the verifier.

Conceptual overview: folding as deferral of proof checking. The protocol of the previous section can be thought of as an argument system for incremental computation that works by reducing the checking of *all* applications of F (or more precisely, of R1CS instances equivalent to F) to checking a single derived folding of the applications of F. That is, the validity of the single folded instance is equivalent to the validity of every one of the applications of F that the prover claims to have faithfully executed.

With this in mind, the (validity of) the running folded committed-R1CS-with-slack-vector instance I at the start of each "round" $j > 1$ itself acts a "proof" π_j that $F^{(j-1)}(x) = y_{j-1}$. The folding procedure that occurs in "round" $j > 1$ should then be thought of as a way to *defer* checking the validity of π_j to a later point. Moreover, the folding has the effect of "accumulating" all i such checks into a single statement that can be checked at the same cost as performing any one of the validations individually. Specifically, the checks are deferred until all i foldings have occurred, at which point the prover finally establishes that the final running folded instance is valid.

The above method of "deferring/accumulating" the checking of each "proof" π_j is in contrast to the recursive-composition-of-SNARKs approach covered in Section 18.4, in which the prover explicitly proves that it verified a SNARK proof π_j in its own head for all $j = 1, \ldots, i-1$.[23] Intuitively, it is cheaper to defer/accumulate the checks than it is to

[23] More precisely, the prover establishes that it knows a π_j that would have convinced the SNARK verifier to accept. But this effectively means that the prover has itself applied the SNARK verifier's accept/reject computation to π_j, since the prover knows that the outcome of this computation is "accept".

actually explicitly perform each check, thereby reducing the overhead of recursion relative to the recursive-composition-of-SNARKs approach of Section 18.4 (we discuss exactly what is the overhead of recursion of Nova later).[24]

The augmented function F'. Now we come to obtaining a SNARK from the folding scheme via recursive proof composition. Let us "augment" the computation of F to a larger computation F' that not only 1) applies F but also 2) does the verifier's work in one step of the folding scheme. This is analogous to how, in Section 18.4, the honest prover in round j of proof generation applied a base SNARK $\mathcal{O}$ to a circuit $\mathcal{C}'$ that not only applied F to y_{j-1}, but also applied a verification circuit to the proof π_{j-1} computed in the previous round $j-1$.

In more detail, F' will take as public input values for the variables maintained by the verifier in round j of the folding scheme (see the bulleted list in Section 18.5.3), and will also take as non-deterministic input the prover's message in the folding scheme (except for the claimed value of y_j). F' will output the new values of the verifier's variables in the folding scheme upon processing the prover's message—see the paragraph entitled "How the verifier processes round j" of Section 18.5.3. The one exception is that whereas the verifier in the folding scheme updates the value of the variable *cur-output* to a *claimed* value for y_j provided by the prover, F' will instead output the *actual* value of y_j. That is, F' will apply F to the relevant input and include the result in its output.

The SNARK. The final SNARK applies the folding-based proof of the previous section with F' in place of F.[25] But rather than outputting

[24] The deferral/accumulation of these checks is also analogous to earlier results such as Halo [75], that deferred/accumulated only *part* of the verification of the SNARK proof π_j, namely the verification of evaluations of committed polynomials, via a folding-like procedure.

[25] This description elides the following subtlety, which requires a tweak to the definition of F' to address. The folding-scheme is applied to force the prover to faithfully compute $(F')^{(i)}$, which means that for $j \leq i$, the output of the $(j-1)$'st application of F' has to be fed as public input to the j'th application of F'. One "piece" of the output of F' is the folding-verifier's variable s representing a "running

the entire proof, which consists of i "rounds", the final SNARK proof provides only the information sent by the prover in the final "round". This information comprises the following:

- A specification of the running folded instance I at the start of round i (Equation (18.15)), and a description of the final R1CS instance I_j to be folded in (Equation (18.16) with $j = i$). This latter description includes the claimed output of $(F')^{(i)}(x)$. This includes both the variable *round-count* (Section 18.5.3) and the claimed output y_i of $F^{(i)}(x)$. The SNARK verifier must confirm that *round-count*$= i$ and reject if not, as this ensures that the proof actually refers to $F^{(i)}$ and not $F^{(j)}$ for some $j \neq i$. If all of the SNARK verifier's remaining checks (described below) pass, then the verifier is convinced that indeed $y_i = F^{(i)}(x)$.

- The information provided by the prover in the "final round" of the protocol of the previous section to perform the final folding operation, specifically a commitment c_T to the cross-term used in this folding operation.

- A SNARK proof that the final folded instance is satisfiable.

In summary, the honest prover performs each "round" of the previous section's protocol in its own head, only outputting a transcript of the

folding" of all public inputs to previous applications of F'. This means that the vector s that is (just one piece of the) input to the jth application of F' has to be at least as big as the entire public input to the previous application of F'. But since there are other outputs of the $(j-1)$'st application of F' as well (see the bulleted list of verifier values in Section 18.5.3), this forces the length of the public input to the jth application of F' to be *strictly bigger* than that of the previous application. Thus, the public input length for F' grows with each application of F'. To address this issue, one can modify F' to not include in its output $s \in \mathbb{F}^m$, but only a cryptographic hash $H(s)$, thereby ensuring that the output length of F' is independent of the length of s. F' will then take s as an additional non-deterministic input rather than as public input and as part of its computation it will confirm that s is indeed the pre-image of the associated public input value $H(s)$. In summary, without this modification, the public input size to F' grows iteration-by-iteration, because the vector s (the folding of prior public inputs) grows with each iteration. The modification replaces s in the input and output of F' with a hash $H(s)$, which addresses the issue because the size of the hash $H(s)$ does not depend on the length of the vector s.

final "round" of the protocol. This is analogous to how the prover in the recursive-SNARK solution of Section 18.4 for $F^{(i)}$ generated in its own head a sequence of SNARK proofs $\pi_1, \ldots, \pi_i$, with each π_j attesting to a correct execution of F to input y_{j-1} (as well as knowledge of π_{j-1}). But ultimately, only the final proof π_i needs to be sent to the verifier to guarantee the correctness of the claimed output of $F^{(i)}$.

Essentially, each time that the Nova prover $\mathcal{P}$ performs a folding operation in its own head, thereby folding I_j into the running folded instance I, the very next application of F' performs the verifier's work in the folding operation, in addition to applying F for a $(j+1)$'st time. This is the sense in which the final Nova SNARK forces the prover of the previous section's protocol to perform in its own head the verifier's work of that protocol.

The overhead of recursion. In this SNARK, the overhead of recursion refers to the amount of extra work that F' does beyond simply applying F (or more precisely, the number of constraints in the R1CS instance over field $\mathbb{F}_p$ representing F' relative to the R1CS instance over $\mathbb{F}_p$ representing F). This extra work done in F' simply implements the verifier's variable updates in the folding scheme; see the final paragraph of Section 18.5.3. This consists of a handful of field multiplications and additions over $\mathbb{F}_p$, one invocation of a cryptographic hash function per the Fiat-Shamir transformation, and the homomorphic updating of the two commitments c_w and c_E to obtain commitments to $w + rw_j$ (as per Equation (18.6)), and $E + rT$.

If a SNARK-friendly hash function is used for Fiat-Shamir, then it is the two homomorphic commitment updates that dominate the overhead of recursion. If the commitments are Pedersen vector commitments over a multiplicative group $\mathbb{G}$, then each of these updates requires one group exponentiation and one group multiplication; it is the two group exponentiations that dominate the cost, as a group exponentiation takes approximately $\log|\mathbb{G}| \approx 2\lambda$ group multiplications. This overhead of recursion is concretely cheaper than that of recursive-SNARK solutions considered earlier in this section (see Footnote 13 in Section 18.4).[26]

[26] This description elides an important implementation issue that is essentially identical to the one described in Footnote 9 in the context of IVC from recursive

Overall prover runtime. Assuming the number of iterations i is not very small, the prover's runtime is dominated by the cost of computing a Pedersen vector commitment at every iteration $j \leq i$ to the witness vector w_j and the cross-term T. Both of these vectors have length at most n', where n' is the number of rows of the R1CS instance capturing F'. Hence, this is two multi-exponentiations of size n' per iteration. As per the above overhead-of-recursion analysis, n' is quite close to n, the number of rows of the R1CS capturing F alone. One does need to use a cycle of elliptic curves, but the curves need not be pairing friendly, ensuring fast group operations (see Section 18.2).

SNARKs. Specifically, Pedersen vector commitments that are homomorphic over field $\mathbb{F}_p$ are elements of an elliptic curve group $\mathbb{G}$ in which the *scalar field* is $\mathbb{F}_p$ and the *base field* is another field, $\mathbb{F}$. And group operations over $\mathbb{G}$ can be efficiently implemented by a circuit or R1CS defined over the base field, but unfortunately not the scalar field. Similar to Footnote 9, one way to sidestep this issue is to identify a cycle of curves $\mathbb{G}$ and $\mathbb{G}'$ with scalar and base fields $\mathbb{F}_p$ and $\mathbb{F}$ such that F is efficiently computable by circuits or R1CS over *both* $\mathbb{F}_p$ and $\mathbb{F}$. One then maintains two different sequences of R1CS instances, with one sequence defined over field $\mathbb{F}_p$ and the other defined over field $\mathbb{F}$. Since F is efficiently computable in R1CS over both fields, one can efficiently define two different augmented functions, say, F', and F'', computing F and performing folding operations when commitments are sent over $\mathbb{G}'$ and $\mathbb{G}$ respectively. One then alternates performing folding operations on each sequence. Specifically, a folding of two committed-R1CS-with-slack-vector instances defined over $\mathbb{F}_p$ (and associated application of F') can be efficiently computed by F'' and hence by the R1CS sequence defined over field $\mathbb{F}$, and similarly a folding operation of two instances defined over $\mathbb{F}$ can be efficiently computed by F' and hence by the R1CS sequence defined over $\mathbb{F}_p$. The final SNARK proof consists of the final folding operation for *both* sequences, and SNARK proofs for both sequences that the final folded instance is satisfied. Also similar to Footnote 9, if F is only efficiently implementable over $\mathbb{F}_p$ one will still have two functions F' and F'', but only F' will both apply F and implement folding; F'' will only implement folding. This will double the number of folding operations required to obtain a SNARK for $F^{(i)}$. Effectively only applications of F' perform the "useful work" of applying F; applications of F'' are only used to "switch" which of the two fields folding operations can be efficiently computed over.