# ASCrypto 2025: Recursive Proofs and Accumulation

September 29 of 2025

Author Emanuel Nicolás Herrador
Speaker Benedikt Bünz

**TODO:** Add scheme figures from slides to make explanations clear.

## 1 SNARKs review

We'll use SNARKs to construct our schemes here, so it's important to have this concepts in mind.

The problem to solve is: given an $x$, exists a $w$ that makes the output true? The prover want to prove that he knows the solution. Here we'll working with non-interactive proofs when the proof is $\pi$ and we want $|\pi| << |w|$ and $|\mathcal{V}_{\text{arg}}| << |w|$ (succinct) because otherwise we can send just the witness and it works (with completeness and knowledge-soundness). Then, verifier outputs 0 or 1 if it's true.

The properties we want are:

- Completeness: if $(x, w) \in R$ then $\mathcal{V}_{\text{arg}} \to 1$
- Soundness: if $x \notin \mathcal{L}(R)$ then with high probability $\mathcal{V}_{\text{arg}} \to 0$
- Knowledge-soundness (more general): we want to find the extractor that can access the prover and compute the witness ($w$). This is important in sites where the soundness is obvious.

  An example of that can be is we want to prove if we know a hash collision. Here there is a big difference between knowing existence and knowing the particular collision.

## 2 Recursive proofs

Consider we've two SKARK systems $(\mathcal{P}, \mathcal{V}), (\mathcal{P}', \mathcal{V}')$ (sometimes the same). Then, a recursive proof is when $\mathcal{P}'$ proves that $\mathcal{V}$ accepted the pair $(x, \pi)$ (instance, proof) from $\mathcal{P}$. After that, it creates a proof $\pi'$ that must be verified by $\mathcal{V}'$.

Formally, $\mathcal{P}'$ proves that it knows a proof $\pi$ for a statement $x$ in a language indexed by a verification key $vk$ such that $\mathcal{V}$ accepts $\pi$ for $(x, vk)$ Knowledge-soundness of $(\mathcal{P}', \mathcal{V}')$ implies we can extract $\pi$. We also says $\mathcal{V}$ as the recursive circuit.

In the following, we're showing some motivations about recursive proofs use and the cons of naive solutions.

**Motivation 1: Prove sequential computations** Here, we've a function $F$ that its applied $t$ times to our instance. Then, we can break it in parts and each one has an specific witness. The definition for an intermediate instance is $x_t = F^t(x_0; w_1, \ldots, w_t)$ and we want to prove that we know this $w_1, \ldots, w_t$.

We can made a *naive solution* using SNARKs as a monolithic proof (1 proof). The instance por $\mathcal{P}$ will be $x_t$ and then it give us a proof $\pi$. I.e., instead of dividing $F$ computation $t$ times, we group all the instance together and therefore in this scheme we can avoid proof-size linearity in $t$.

The issues of this solution is that it's not memory-efficient (because I need to store all the intermediate computation in memory to create a proof). Also, sometimes the prover is super-linear in $t \cdot |F|$. And the biggest issue is that additional steps requires reproving everything (it's bad because we don't save intermediate proofs to use it later, we should here recompute the entire history proof). Then, the proofs are linear in the entire history of the computation and we don't want it.

**Motivation 2: Handing off computation** It's a similar description as before but here it isn't a line and each party makes the computation apart. Here, each part wants to verify the inputs and some wants to check the entire computation. The *naive solution* is creating a proof (each party) and attach them. However it will be linear in the number of steps.

## 2.1 Incrementally verifiable computation (IVC)

It's a way to solve the last problems efficiently avoiding this naive solutions. In each step, when $F$ is using evaluating value $x_i$ then it also creates the instance $x_{i+1}$ and the proof $\pi_{i+1}$. Finally, the last proof $\pi_t$ is sent to the verifier (notice that each step is the same prover $\mathcal{P}$).

We want:

- Completeness: Given valid proof $\pi_{i-1}$ for $x_{i-1}$, $\mathcal{P}$ generates a valid proof $\pi_i$ for $x_i := F(x_{i-1}, w_i)$
- Knowledge-soundness: Given a valid proof $\pi_i$ for $x_i$, extract witnesses $w_1, \ldots, w_t$ such that the following holds: $x_t = F^t(x_0; w_1, \ldots, w_t)$
- Efficiency: Proof-size and prover/verifier-runtime should be independent of $t$

*Remark.* The generalization of IVC is PCD (Proof Carrying Data) to permits arbitrary DAGs structures. It needs upper bound on arity.

**IVC from recursive composition of SNARKs** In each step we want a relation $R$ where we say that $x_{t+1}$ is well constructed from $x_t$ and also $\pi_t$ is verified (a verifier outputs 1). This relationship is constructed as a circuit that will be used for the ICV prover. From $R$ is constructed our SNARK prover $\mathcal{P}$ used by our IVC module $\mathcal{P}_F$ and with our verification key $Vk$.

*Remark.* The circuit has a special case for the first module where isn't necessary a proof (because we don't have it).

The verifier IVC $\mathcal{V}_F$ gets $x_1, x_t$ and $\pi_t$ and should output the verification result. I need the first and last step of the computation.

For the properties, we've that:

- Completeness: Follows from SNARK completeness
- Soundness: Recusively extract transcript using SNARK knowledge-soundness (using the verifier parameters, we can reconstruct the instance and proofs used in steps before made)
- Efficiency: $|\pi|$ is the size of a SNARK proof for $R$. We want that the size of the SNARK proof doesn't grow and. Then, it's necessary sublinear verification to bound $R$ to avoid growing up each time we construct a new proof instance.

Notice that we're using SNARKs to avoid interactivity because we can't construct our $R$ circuits with interactivity.

We can see some applications in the follow examples.

**Application 3: Property preserving SNARKs** The goal here is to improve SNARK prover properties. We've a SNARK $A$ that has a fast sequential prover (non parallel) with large memory, CRS and verifier. We want to convert it to a new SNARK $A'$ with a fast parallel prover with constant memory, CRS and verifier-size.

The solution is to break up function $F$ into $T$ uniform steps $F'$ of size $\frac{|F|}{T}$.

**Application 4: SNARK composition** The goal is combining SNARKs with different tradeoffs. We've SNARK A (fast prover, slow verifier, large proofs) and SNARK B (slow prover, fast verifier, small proofs, ZK) and we want a SNARK C (fast prover and verifier, small proofs, ZK).

The solution is to use SNARK B to prove correctness of SNARK A.

**Many more applications** There're a lot of other applications like image provenance, verifiable delay functions, succinct blockchains, ZK cluster computing, signature aggregation, and more.

Recursive proofs are widely deployed in real world (succinct, zsSync, polygon, Nexus, etc.).

## 2.2 Security issues for IVC

**Arithmetizing $\mathcal{V}$** The problem is that as $R$ contains $\mathcal{V}$, then $\mathcal{V}$ can't contains oracles because we need to implement $\mathcal{V}$ as a circuit.

For that, we've to have in mind the following. The *security jump* is purely heuristic (there isn't a proof of that and also it is generally not true) and it says that if we've $(\mathcal{P}^\rho, \mathcal{V}^\rho)$ secure in RO then $(\mathcal{V}, \mathcal{P})$ = *Fiat-Shamir*$(\mathcal{P}^\rho, \mathcal{V}^\rho)$ is secure in the standard CRS model. Also, there're a recent attack on GKR that relies on evaluating FS-Hash inside proofs systems but recursion relies on this ability.

So, we can't have $\mathcal{V}$ containing an oracle because it's insecure and there exists attacks for that. The model isn't broken, the real world deployment is broken for this heuristic jump.