# Constructing hard instances in TFNP subclasses relative to a random oracle

January 26, 2025

## List of papers considered:

- IVC and its variants
  - ☑ [Val08]: basic IVC and hardness results.
- CLS hardness on average relative to ROM using (1) non-interactive sumcheck protocol or (2) continuous verifiably delay functions (cVDF).
  - ☑ [MMV11]: In ROM it is impossible to efficiently create a puzzle **while creating a solution**, such that in the mean time there is a super-linear gap between solving the puzzle and verifying the puzzle.
  - ☑ [MMV13]: In ROM, we have CRHFs and $\tau$-sequential functions. Using that on top of an explicitly constructible depth-robust DAGs, they construct time-lock puzzles with super-polynomial gap between solving a puzzle and verifying a solution. These time-lock puzzles are used to construct publicly verifiable PoSW.
  - ☑ [CP18]: CP graph construction of PoSW. It simplifies the proof of [MMV13] and got rid of the dependency on DRG.
  - ☒ [DLM19]: Is a follow-up work to [CP18] to support incremental proofs (proof for $N$ steps of work and another proof on top of that for $N'$ steps of work prove, together, a proof of $(N + N')$ steps of work). Not elaboated in much technical details
  - ☑ [CHK$^+$19]: Computational uniqueness (as compared to actual uniqueness) of IVC is enough to build hard CLS instances; CLS hardness in ROM, but need the additional assumption that #SAT is hard. CLS hardness $\implies$ PPA hardness.
  - ☑ [EFKP20]: CLS is hard using repeated squaring protocol assuming repeated squaring modulo a composite is hard to parallelize. The exact instance constructed can be solved in some tunable poly-time, sequentially, but parallelizing cannot improve this run-time by much.
- ☑ PLS hardness on average relative to ROM: it is a higher class so should have simpler proofs with potentially weaker assumptions than CLS:
  - ☑ [BG20] PLS hardness from CP graph and computationally sound IVC. It uses properties of such construction in ROM.

## Summary

| Citation | Described | Assumptions | Hardness achieved | Proof Strategy |
|---|---|---|---|---|
| [CHK+19] | Sumcheck | $\#SAT$ is HOA in $\mathsf{ROM}$ | $\mathsf{rSVL}^{\mathcal{O}}$ HOA instance | Unambiguity |
| [EFKP20] | Repeated squaring | Repeated squaring assumption (RSW, which implies FACTORING is hard) | $\mathsf{rSVL}^{\mathcal{O}}$ HOA instance | Uniqueness |
| [BG20] | iPoSW | Unconditional in $\mathsf{ROM}$ | $\mathsf{PLS}^{\mathcal{O}}$ HOA instance | No uniqueness |

The construction of all of these papers are more or less based on the initial tree construction of [Val08] which uses a "strong" knowledge extractor as a way to merge proofs efficiently. The point is [Val08] does not guarantee uniqueness, and so not sufficient to construct a $\mathsf{rSVL}$ instance.

# Contents

**1 Overview** **5**

**2 About** $\mathsf{ROM}$ **5**

2.1 What can we NOT DO in $\mathsf{ROM}$ . . . 5

2.2 What can we DO in $\mathsf{ROM}$ . . . 5

2.2.1 Random oracle is collision-resistant . . . 5

2.2.2 Random oracle is sequential . . . 6

2.2.3 Combining collision-resistance with sequentiality . . . 7

2.2.4 Fiat-Shamir heuristic . . . 8

**3 IVC construction #1 in** $\mathsf{ROM}$ **unconditionally without uniqueness [Val08]** **8**

3.1 Where $\mathsf{ROM}$ fits? . . . 9

3.2 Outline . . . 9

3.3 Merging proofs . . . 9

3.3.1 Base case: A member of $\mathcal{L}_c$ that proves one time step . . . 10

3.3.2 Inductive case: To generalize to a member of $\mathcal{L}_c$ that proves increasingly many steps . 10

3.3.3 Computationally sound: . . . 11

3.4 Main theorem . . . 11

3.5 Constructing the required noninteractive CS PoK in $\mathsf{ROM}$ [Val08] . . . 13

3.5.1 Witness-extractable PCPs . . . 14

3.5.2 CS proof construction via Merkle trees in $\mathsf{ROM}$ . . . 14

3.5.3 Verification and knowledge extraction . . . 14

**4 Why is uniqueness / unambiguity useful?** **15**

**5 IVC construction #2 in** $\mathsf{ROM}$ **assuming #SAT-hardness with unambiguity [CHK+19]** **15**

5.1 Interactive $\beta$-prefix sumcheck protocol for $y = \sum_{z_{j+1},\ldots,z_n \in \mathbb{H}} f(\beta_1, \ldots, \beta_j, z_{j+1}, \ldots, z_n)$ . . . 15

5.1.1 Unambiguous interactive proof protocol . . . 16

5.2 Explicit construction of an unambiguously sound IP for a particular language . . . 16

5.3 #SAT $\leq$ $\mathsf{rSVL}$ . . . 21

5.3.1 Arithmetization of SAT . . . 21

5.3.2 The reduction (set-up and statement) . . . 21

5.3.3 The successor circuit, $S = S_n$ . . . 22

5.3.4 The verification circuit, $V = V_n$ . . . 24

5.3.5 Hardness of this $\mathsf{rSVL}$ instance . . . 26

5.4 $\mathsf{rSVL} \leq \mathsf{EoML}$ and $\mathsf{EoML} \in \mathsf{CLS}$ . . . 27

5.5 Highlights . . . 29

**6 IVC construction #3 in** $\mathsf{ROM}$ **assuming iterated squaring is hard and inherent sequentiality (in turn factoring must be hard) with uniqueness [EFKP20]** **29**

6.1 Verifiable, sequential and iteratively sequential functions . . . 29

6.2 Repeated squaring assumption (aka the RSW assumption for Rivest-Shamir-Wagner, originally used to construct time-lock puzzles) . . . 30

6.3 A failed attempt . . . 31

6.4 cVDF as modified from the Pietrzak's protocol construction . . . 31

6.4.1 Construction of cVDF . . . 31

6.4.2 Strengthened version of $\mathcal{L}_{N,B}$ to ensure adaptive soundness, efficient verification, and completeness . . . 32

6.4.3 Adaptive soundness intuition . . . 32

6.4.4 Interactive proof protocol for $\mathcal{L}_{N,B}$ and uVDF as a result . . . 33

6.4.5 Claims about the constructed uVDF . . . 34
6.4.6 cVDF from uVDF . . . 35
6.5 Use cVDF to construct hard-on-average $\mathsf{rSVL}$ . . . 37

**7 IVC construction #4 in $\mathsf{ROM}$ assuming unconditionally without uniqueness [CP18, DLM19, BG20] 38**
7.1 What did [MMV13] do? . . . 39
7.2 The CP graph . . . 39
7.3 PoSW based on the CP graph . . . 40
7.3.1 Construction . . . 41
7.3.2 Non-interactiveness in $\mathsf{ROM}$ . . . 42
7.4 Proof of security . . . 42
7.5 Efficiency . . . 42
7.5.1 Prover efficiency . . . 43
7.5.2 Verifier efficiency . . . 43
7.6 Lack of uniqueness property . . . 43
7.7 Construction tweaks . . . 43
7.7.1 Why can't [CP18] construction be incremental? . . . 43
7.7.2 Construction . . . 44
7.8 Parameter improvements . . . 45
7.9 Construction of hard-on-average $\mathsf{PLS}^{\mathcal{O}}$ instances . . . 45
7.9.1 Construction overview . . . 46
7.9.2 DLM iPoSW modified . . . 46
7.9.3 Properties of the modified DLM construction . . . 48
7.9.4 Hard $\mathsf{PLS}^{\mathcal{O}}$ instance, $R_d^{\mathcal{O}}$ . . . 48

**8 Directions 49**

## 1 Overview

We first give a summary of what's so useful about being in $\mathsf{ROM}$. Then, we see some constructions of interactive proof systems (made non-interactive in $\mathsf{ROM}$) with varied assumptions.

- Some of these constructions use assumptions to guarantee some level of uniqueness (in the sense that prover must follow a certain proof strategy; see sections 5 - 6). With uniqueness (and unambiguity as a weaker property), they can be used to construct hard instances of $\mathsf{rSVL}^{\mathcal{O}}$.
- Others are unconditional in $\mathsf{ROM}$ but do not have the uniqueness property, like the earliest *incrementally verifiable computation* (IVC) construction in section 3 and the construction used to show a hard $\mathsf{PLS}^{\mathcal{O}}$ instance in section 7.

## 2 About ROM

Here, we summarize some basic properties that you DO have and DON'T have in $\mathsf{ROM}$.

### 2.1 What can we NOT DO in ROM

[MMV11] proved some impossibility results regarding time-lock puzzles in $\mathsf{ROM}$.

**Theorem 1** (Optimally adaptive but inefficient adversary)**.** *Suppose a time-lock puzzle generator with access to a cryptographic hash function (available in* $\mathsf{ROM}$*)* $f^H(r_A)$ *uses* $n$ *queries to the random oracle* $H$ *and generates* ***a puzzle*** $M$ ***and a verifier*** $V$*. Suppose also there is a solver* $g$ *that sulves the puzzle with* $m > n$ *queries such that the success probability is at least* $1 - \epsilon$*. Then, there exists an adversary who:*

1. *Uses* $n$ *rounds of adaptivity.*
2. *Makes* $\tilde{O}(n \cdot m/\epsilon')$ *queries to* $H$*.*
3. *Solves the puzzle with probability* $\geq 1 - \epsilon - \epsilon'$*.*

**Theorem 2** (Non-optimally adaptive but efficient adversary)**.** *With the same set-up as the previous theorem, but in this case the adversary makes more non-adaptive calls to run in less time:*

1. *Uses* $n/\epsilon'$ *rounds of adaptivity.*
2. *Makes* $n \cdot m/\epsilon'$ *queries to* $H$*.*
3. *Solves the puzzle with probability* $\geq 1 - \epsilon - \epsilon'$*.*

**Remark 3.** *Note what this gap here is saying: the work required to generate both* ***a puzzle and a verifier / solver*** *can be at most linearly different from the time required for the adversary to solve the same puzzle with high probability, adaptively or non-adaptively, with high probability. Why is work in this line of work still possible? It turns out ok if the puzzle generator only computes the puzzle (i.e. not the solution at the same time).*

Why is this relevant? Turns out time-lock puzzles can be used to construct publicly verifiable PoSW in $\mathsf{ROM}$, which is the focus of the follow-up work [MMV13] summarized next.

### 2.2 What can we DO in ROM

#### 2.2.1 Random oracle is collision-resistant

Collision resistance is when it is hard to find two inputs to a function that are mapped to the same output. In $\mathsf{ROM}$, we have the following lemma:

**Lemma 4.** *Suppose any $\mathcal{A}^H$, which is a PPT with access to a random function*

$$H : \{0,1\}^* \to \{0,1\}^w,$$

*If $\mathcal{A}$ makes at most $q$ queries,*

$$\Pr[x \neq x' \textit{ be two of the queries } \mathcal{A} \textit{ makes}, H(x) = H(x')] \leq \frac{q^2}{2^{w+1}}.$$

*In particular, $\frac{q^2}{2^{w+1}}$ is negligible so by definition of collision-resistance we have $H$ that is collision-resistant in* ROM.

*Proof.* Recall that $H$ is uniformly random. In the worst case, $\mathcal{A}^H$ can try $q$ different values of $x$. Then,

$$\Pr[x \neq x' \text{ be two of the queries } \mathcal{A} \text{ makes}, H(x) = H(x')] \leq \sum_{i \in [2,q]} \frac{i-1}{2^w},$$

by union bound on the probability that the hash value of the $i$-th query turns out to collide with the hash value of one of the previous queries. This is just

$$\sum_{i \in [1,q-1]} \frac{i}{2^w} = \frac{q(q-1)}{2^{w+1}} \leq \frac{q^2}{2^{w+1}}.$$

■

### 2.2.2 Random oracle is sequential

Inherently sequential hash functions (a new device of [MMV13]'s discovery). The idea is, no matter how much pre-processing an adversary makes (consequently the ability to make many meaningful parallel queries), the number of adaptive queries that the adversary is required to make cannot be considerably improved.

**Lemma 5** (RO is sequential)**.** *Suppose any $\mathcal{A}^H$, which is a PPT with access to a random function*

$$H : \{0,1\}^* \to \{0,1\}^w,$$

*that $\mathcal{A}$ can query for at most $(s-1)$ rounds, though in each round it can make arbitrarily many parallel queries. There exists an output sequence that $\mathcal{A}$ can construct using $q$ queries (totaling a length of $Q$ bits) with probability only*

$$\leq q \cdot \frac{Q + \sum_{i \in [0,s]} |x_i|}{2^w},$$

*where $q_i$ is the $i$-th query string.*

*Proof.* This output sequence can be made explicit, it is

$$\{x_0, \ldots, x_s\}, \text{ where } \exists a, b \in \{0,1\}^*, x_{i+1} = a \| H(x_i) \| b.$$

We call such a sequence an $H$-sequence.

For the $H$-sequence to be constructed in $q < (s-1)$ rounds, an adversary must be able to do one of the following things: (1). it holds $x_{i+1}$ and guesses a correct $x_i$ such that $H(x_i)$ is a substring of $x_{i+1}$; (2). let $i \leq j \in [1, s-1]$, such that $a_i$ is a query made first and $a_j$ a query that is made next that just turned out $H(a_j)$ is a substring of $a_i$. We bound these respectively:

- First case, which is a lucky guess, happens only if part of a query guessed $H(x_i)$ correctly, which has a length of $w$ by definition of the hash function, so union bound over $q$ queries, we have

$$\Pr[\text{Right guess in any query}] \leq \sum_{i \in [0,s]} q \cdot \frac{|x_i|}{2^w},$$

where $q_i$ is the $i$-th query string.

- In the second case, there is at most a $\frac{|a_i|}{2^w}$ chance for it to happen to collide with any query, since $H$ is uniformly random, meaning that colliding with a length $|a_i|$ query after the hash is a uniform probability as written. Union bound over all queries, we have $\sum_{i \in [1,s-1]} |a_i| \leq Q$, so

$$\Pr[\text{Nonseuqntual queries collide}] \leq q \cdot \frac{Q}{2^w}.$$

Adding these two probabilities gives the bound claimed in the lemma. ■

**Amplification.** The traditional Merkle-Damgard construction allows us to polynomially amplify the scale of shrinkage of a hash function. It is shown that this construction can be modified so that we can amplify a $\tau$-sequential family of hash functions to some other family of hash functions that is also sequential but with more shrinkage power. The exact process is as follows:

1. Take $H_1 = \{h_s : \{0,1\}^{m_1(n)} \to \{0,1\}^n\}$ be a family of $\tau$-sequential hash functions (suppose $m_1 = c \cdot n$, where $c > 1$). Let $h_s^{(1)} \in H_1$.

2. Let $m_2(n) = poly(n)$. Suppose an input of length $m_2$, i.e. $(x_1, \ldots, x_{m_2})$. Then, we apply $h_s^{(1)}$ as follows:
$$y_1 = h_s^{(1)}(x_1, \ldots, x_{m_1}), \ldots, y_i = h_s^{(1)}(x_{(i-1)(m_1-n)+1}, \ldots, x_{(i-1)(m_1-n)+m_1}), \ldots$$

3. We continue shrinkage in the above manner as a Merkle tree until the length becomes $m_1$, at which point we take one more hash to get an output of length $n$, call it $z$. We output $z$.

This process gives us a family of hash functions based on $H_1$:

$$H_2 = \{h_s : \{0,1\}^{m_2(n)} \to \{0,1\}^n\}.$$

**Lemma 6.** *$H_2$ is $\Omega(\tau \cdot \log_\rho n)$-sequential. This means that the sequential guarantee decays by a factor of $\log(n)$.*

Proof in [MMV13], lemma 3.9.

### 2.2.3 Combining collision-resistance with sequentiality

Let $\mathbf{D}$ be the distribution of documents. Let $s \overset{\$}{\leftarrow} \mathbf{D}$. Suppose we have:

- $CH = \{h_s : \{0,1\}^m \to \{0,1\}^{n_1}\}$ which is a family of collision resistant hash functions.
- $SH = \{h_s : \{0,1\}^m \to \{0,1\}^{n_2}\}$ which is a family of sequential functions.

**Lemma 7.** *We can use $CH$ and $SH$ to construct a family of hash functions:*

$$H = \{h_s : \{0,1\}^m \to \{0,1\}^{n_1+n_2}\},$$

*which is both collision-resistant and sequential w.r.t. $s \overset{\$}{\leftarrow} \mathbf{D}$.*

*Proof.* Here's the construction: Let $s$ be sampled the way described, so $ch_s \in CH$ and $sh_s \in SH$. Then

$$h_s(x) = ch_s(x) \circ sh_s(x) \in H.$$

- Collision-resistant: suppose we find $x \neq x'$ such that $sh_s(x) \circ ch_s(x) = sh_s(x') \circ ch_s(x')$. Clearly, we can take the first $n_1$ bits of the result and break the collision-resistance of $ch_s$.
- Sequentiality: suppose $h_s(x_i)$ is a contiguous substring of $x_{i+1}$. Then, $sh_s(x)$ which is a contiguous substring of $h_s(x_i)$ must in turn be a contiguous substring of $x_{i+1}$. So if an adversary breaks $H$'s sequentiality with non-negligible probability, it also breaks $SH$.

■

### 2.2.4 Fiat-Shamir heuristic

**Definition 8** (Argument)**.** *A non-interactive proof system that has soundness and unambiguity properties against computationally-bounded cheating prover strategy $\tilde{\mathcal{P}}$ is called an argument.*

Fiat-Shamir heuristic is one way to convert a protocol that has many rounds of interactions into an argument with a single message. It is commonly phrased in the $\mathsf{ROM}$ with CRS, in which case the verifier's challenges are uniformly randomly sampled. Then, instead of receiving these messages from the verifier, the prover can generate these challenges by hashing its transcript, using a strong hash function that the prover and the verifier share.

**Example 9.** *Suppose in $\mathsf{ROM}$ with CRS where the $\mathcal{P}$ first sends $\mathcal{V}$ $\alpha$, and $\mathcal{V}$ sends its randomly sampled challenge $\beta$, to which $\mathcal{P}$ has to respond $\gamma$ to convince the verifier.*

*Now, instead of the CRS, suppose both $\mathcal{P}$ and $\mathcal{V}$ share access to the same hash function $h \leftarrow \mathcal{H}$. Then, the prover can instead send a single message:*

$$(\alpha, h(\alpha), \gamma).$$

**Remark 10.** *Soundness of the Fiat-Shamir transformation has been disproved for contrived protocols, but its soundness is not known when applied to a natural protocol.*

## 3 IVC construction #1 in ROM unconditionally without uniqueness [Val08]

The common strategy to show $\mathsf{rSVL}^{\mathcal{O}}$ is HOA in $\mathsf{ROM}$ is by constructing proof systems that are incrementally verifiable without blowing up the proof size + have uniqueness / computational ambiguity. So, before constructions in sections 5 and 6 that are actually used to show $\mathsf{rSVL}^{\mathcal{O}}$ hardness, we see the preceding construction by [Val08].

The setting is for $M$ to be compiled into $M'$ such that $M'$ outputs $(c_i, \pi_i)$ at various time points where $c_i$ is the configuration of $M$ at $i$-th time and $\pi_i$ is the proof of its correctness (so that adversary $M'$ cannot just output random stuff that cannot be verified). How to compile such that $M'$ uses about as much resource as $M$? Consider $t$ is exponential in $k$:

- Naive approach: $M'$ simulates $M$ and records all the memory states of $M$, which is used to output a proof for the correctness of $c_i$. But this proof has a size of $tk$, so it will be $tk$-time and $tk$-space for the verification of the proof. Both are exponential in $k$.
- The use of PCPs improves this process to poly-time in $k$ for verification, but exponential time in $k$ to construct and transmit the proof.

- Computationally sound proofs further improves the length of the transmitted proof from exponential to polynomial in $k$. But as it requires a time interval of at least $t$ between consecutive proofs, the memory requirement is $t = \exp(k)$.

**Remark 11.** *This line of relaxation is meaningful, because a succinct proof that is unconditionally sound would imply* $\mathbf{NP} \subseteq Co\mathbf{NP}$*, which is unlikely.*

The goal is to not blow up the memory size in the context of CS proofs. This can be done if an efficient way of merging two CS proofs of equal lengths into a single CS proof can be found such that the merged proof is short and efficient to verify just like any of the two original proofs. It is not known if this ideal goal can be achieved, but [Val08] showed this for CS proofs of knowledge (which is a variant of the original CS proofs). Intuitively, it is a proof for "$A$ is convinced that $B$ has seen $w$ such that $R(x, w) = 1$", rather than a proof for "$\exists w$ such that $R(x, w) = 1$". The problem is that, by the property of CS proofs, if $x$ is false, a CS proof of $x$ still exists.

Turns out the technical details below are to show that the following merged proof is possible: "$A$ is convinced that $B$ *probably* has seen $w$ such that $R(x, w) = 1$". This is because proof of knowledge can be combined such a way, and the proof is more along the line of "Prover $A$ has seen a computationally sound proof that Prover $B$ has seen a witness $w$ of $x$". A bit more concretely, [Val08] uses ideas of typical ZK proofs of knowledge to construct proofs, in the context of giving short, efficiently verifiable CS proofs (in contrast to exact proofs) for each next time step, as compared to preserving privacy in the ZK context.

## 3.1 Where ROM fits?

Embedding proof systems does not allow what's being proved to have access to the random oracle. Consider an oracle-based prover-verifier system $(P^O, V^O)$. But, instead of proving "$M$ accepts this string within $t$ time steps...", it is really proving "$M^O$ accepts this string within $t$ time steps...", which is a statement about classical computation anymore.

It is however possible to use ROM in other manners, as we will see later.

## 3.2 Outline

Sections 3.3 - 3.5 are some key results of [Val08]. In summary, [Val08] constructs a non-interactive computationally sound proof of knowledge (CS PoK) and uses it to construct a incrementally verifiable computation scheme. Unlike previous proof systems, this new IVC proof system allows the following: efficient proving, efficient verification, short proof size. [Val08] shows that the CS PoK with the required parameters can be constructed in ROM using a Merkle tree. In fact, its framework, which relies on a 3-tuple of (commitment scheme, random challenge, opening scheme) for the prover to prove to the verifier, is seen in later works like [CP18, DLM19, EFKP20, CHK$^+$19].

**(Lack of uniqueness in prover strategy).** The CS proof construction is inherently not unique because it leverages randomness at multiple stages: the underlying witness-extractable PCP proof can be encoded in various valid ways, and the use of a random oracle to generate random queries (which in turn determine the Merkle tree commitment) ensures that even for the same NP witness, different proofs can be produced. This lack of uniqueness is a problem when constructing rSVL instances, which is addressed through a different constructions, namely the ones in [EFKP20, CHK$^+$19].

## 3.3 Merging proofs

**Definition 12** ($\mathcal{L}_c$)**.** *Let $c$ be a constant, the language $\mathcal{L}_c$ consists of the ordered pairs $(M, x)$, where $M$ is a Turing machine and $x$ is a string such that, letting $k = |M|$ we have:*

- $|x| = 3k$.

- *(Short witness + efficiently verifiable). $\exists$a string $w$, with $|w| = 3k$, such that $M(x, w)$ accepts within time $k^c$.*

*Easy to see that $\mathcal{L}_c$ is* **NP***-complete, with $w$ being the witness for $x$ of varying lengths.*

**Definition 13** ($M : s_1 \xrightarrow{t} s_2$)**.** *$M : s_1 \xrightarrow{t} s_2$ is a claim that simulating $M$ starting from the configuration $s_1$ for $t$ time steps, the resulting configuration of $M$ is $s_2$.*

**Theorem 14.** *It is computationally sound to prove the claim "$M : s_1 \xrightarrow{t} s_2$" by proving that $x \in \mathcal{L}_c$, where $x =$ "$M : s_1 \xrightarrow{t} s_2$"*

### 3.3.1 Base case: A member of $\mathcal{L}_c$ that proves one time step

Let $x = (M, s_1, s_2)$ so that $|x| = 3k$. Let $w$ be a trivial placeholder, as it will never be used. $T_0$ is a TM, which is to do the following thing: it simulates $M$ from configuration $s_1$ for one step, and check if it lands in $s_2$. By definition 12, $(T_0, x) \in \mathcal{L}_c$.

### 3.3.2 Inductive case: To generalize to a member of $\mathcal{L}_c$ that proves increasingly many steps

Now, we extend the construction in section 3.3.1 to proofs for longer steps. Firstly, we similarly have $(T_i, s_1, s_2) \in \mathcal{L}_c$ and $(T_i, s_2, s_3) \in \mathcal{L}_c$, which denotes that $T_i$ simulates $2^i$ steps from $s_1$ to get to $s_2$, and from $s_2$ to get to $s_3$. Clearly, $2^i$ is not within $k^c$ time in general, so $w$ will be useful here.

**Definition 15** (Noninteractive CS proof of knowledge)**.** *$(P, U)$ is a noninteractive CS proof of knowledge in the CRS model with parameters $K'(k) : \mathbb{Z}^+ \to \mathbb{Z}^+$, $c, c_1, c_2$ if $P$ and $U$ are TMs, such that for all machines $M$, defining $k = |M|$, and for all string $x$ of length $3k$, there exists a witness $w$, such that the following properties hold:*

- *(Efficient prover). For any CRS $r$ with $|r| = k$, the prover run time on $M, x, w, r$ is poly$(k)$.*
- *(Length shrinking). For any CRS $r$ with $|r| = k$, the size of the proof, which is the output of the prover, is $k$, i.e. $|P(M, x, w, r)| = k$.*
- *(Efficient verification). For any CRS $r$ with $|r| = k$, the verifier runtime on $O(M, x, w, r), M, x, r$ is at most $k^{c-1}$.*
- *(Completeness). For any CRS $r$ with $|r| = k$, $U(P(M, x, w, r), M, x, r) = 1$, i.e. any proof that $P$ provides should be valid for $U$ to accept on the very thing $P$ proves (that said, this is not soundness).*
- *(Knowledge extraction). $\exists c_2$, a constant, such that $\forall$TM $P'$, $\exists$a randomized TM $E_{P'}$, called the extractor, such that $\forall$input $(M, x)$ with $|(M, x)| = 4k, \forall r$ with $|r| = k$, time$_{P'}(M, x, r) \leq K'(k)$ and $\Pr_r[U(P'(M, x, r), M, x, r) = 1] = \alpha > 1/K'$, we have*

  $$\Pr[w \leftarrow E_{P'}(M, x) : M(x, w) = 1] > \frac{1}{2}.$$

  *Furthermore, $E_{P'}(M, x)$ runes in $k^{c_2}/\alpha$ times the expected runtime of $P'(M, x, r)$.*

  *In other words, the knowledge extractor is to produce a witness that will lead $M$ to accept based on $(M, x)$, in some efficient runtime, with majority probability.*

Let $p_1$ and $p_2$ be the CS proofs of knowledge that $(T_i, (M, s_1, s_2)) \in \mathcal{L}_c$ and $(T_i, (M, s_2, s_3)) \in \mathcal{L}_c$, respectively. So, by definition, if we let $x = (M, s_1, s_3)$, $w = (p_1, p_2, s_2)$, and $i$, then $w$ witnesses the fact that $(T_{i+1}, x) \in \mathcal{L}_c$, because verifier can use $p_1$ to efficiently verify $(M, s_1, s_2)$ and $p_2$ to efficiently verify $(M, s_2, s_3)$. Therefore, by definition, we can efficiently construct the CS proof of $x = (M, s_1, s_3)$ by $P(T_{i+1}, x, w, r_{i+1})$, for some $r_{i+1}$ dependent on $i$.

### 3.3.3 Computationally sound:

We prove the above construction, $P(T_{i+1}, x, w, r_{i+1})$, as a CS proof for $(T_{i+1}, x) \in \mathcal{L}_c$, is computationally sound.

**Definition 16** (Deceptive proof tuple)**.** *$(x = (M, s_1, s_2), p)$ is deceptive if $p$ successfully convinces the verifier that "$(T_i, x) \in \mathcal{L}_c$", but "$(T_i, x) \in \mathcal{L}_c$" is not true, i.e. after running $M$ for $2^i$ steps from configuration $s_1$ does not reach $s_2$.*

**Lemma 17.** *For $\alpha \in \left(\frac{1}{K'}, 1\right)$ and $b \in (2(2^i + k), K')$, if $T_i$ has the property that $\nexists$machine that runs in time $b$ outputs deceptive $((M, s_1, s_2), p)$ w.p. $\geq \frac{1}{2}$ over random strings $r_0, \ldots, r_i$, then no machine running in time $\frac{\alpha}{2} b / k^{c_2}$ outputs a deceptive pair for the machine $T_{i+1}$ w.p. $\geq \alpha$ over random strings $r_0, \ldots, r_{i+1}$.*

*Proof.* Proof by induction. The base case: Since the verification for $T_0$ is just running one step, that for $T_1$ is just running two steps and check without having to do with $w$. So it should be true that there should be deceptive proof.

The inductive case: Suppose $P'$ outputs a deceptive pair $((M, s_1, s_3), p')$ for $T_{i+1}$ with probability $\alpha$ over $r_0, \ldots, r_{i+1}$ in time $\frac{\alpha}{2} b / k^{c_2}$. Now consider the $E_{P'}$, which is guaranteed to exist by the knowledge extraction condition of definition 15,

$$\Pr_{\vec{r}}[w \leftarrow E_{P'}(T_{i+1}, x) : M((M, s_1, s_3), w)] \geq \frac{1}{2}, time_{E_{P'}} \leq \frac{b}{2}.$$

Recall that this $w$ is classical, for $(T_{i+1}, (M, s_1, s_3))$ as a member in $L$. We parse $w$ as $(p_1, p_2, s_2)$, such that $p_1$ or $p_2$ or both are deceptive proofs (with $T$ and $r_i$ fixed). Since $b$ time gives enough power to check $2^i + k$ operations, we simply run $M$ on $s_1$ for $2^i$ steps. If the result is not $s_2$, then we know $p_1$ is deceptive. If the result is $s_2$, then we know $p_2$ is deceptive.

So within time, we recovered a deceptive pair for $T_i$ w.p. $\geq \frac{1}{2}$. This contradicts with the assumption on the property that $T_i$ satisifies. ■

**Corollary 18.** *Now, use this lemma, we let $b$ start with $K'$ in the first iteration. Then, for the first $(i-1)$ iterations, we let $\alpha = \frac{1}{2}$. For the last iteration, let $\alpha = \epsilon$. Easy to check, at each step, the condition given in the above lemma is satisfied. The final result would be a time of*

$$K' \frac{1}{(4k^{c_2})^{i-1}} \cdot \left(\frac{\epsilon}{2k^{c_2}}\right) = \frac{2\epsilon K'}{(4k^{c_2})^i}.$$

*Thus, if a machine runs for time $\frac{2\epsilon K'}{(4k^{c_2})^i}$, then it will not output a deceptive pair for $T_i$ w.p. $\geq \epsilon$, over $r_0, \ldots, r_i$. In this case, the proof $P(T_{i+1}, x, w, r_{i+1})$ as a proof for $(T_{i+1}, x) \in \mathcal{L}_c$ is computationally sound.*

## 3.4 Main theorem

**Theorem 19.** *From noninteractive CS proof of knowledge $(P, U, K', c, c_1, c_2)$ to an incrementally verifiable computation scheme $(C, V, K)$*

This is the main theorem of [Val08]. First, we clarify what these tuples mean:

- Definition 15 specified what each of the elements in the tuple means.
- We define IVC $(C, V, K)$. Firstly, we define what a feasible compiler is:

  **Definition 20** (Feasible compiler)**.** *Let $C(\cdot, \cdot)$ be a poly-time TM. $C$ is a feasible compiler if $\exists$a constant $c$ such that $\forall k > 0$, and $\forall M()$ such that $|M| \leq k$, $C(M, k)$ is a Turing machine $T(\cdot)$, taking a CRS $r$ as an input, such that*

- *(Frequency) $T$ is $k^c$-incremental (each pair of neighboring time points it outputs something are different by at most $k^c$).*
- *(Space efficiency) $space_T(r) = k^c$.*

**Definition 21** (IVC, $(C, V, K)$)**.** *$(C, V)$ is said to be an IVC scheme with security $K$ if $C$ is a feasible compiler, $V$ is a poly-time verifier, and $K(k) : \mathbb{Z}^+ \to \mathbb{Z}^+$, such that the following peroperties hold: for any TM $M$ with $|M| \leq k$, let the $j$-th output and proof of $C(M, k)$ is written as $(m_j, \pi_j^r)$ and let $r$ with $|r| = k^2$ be the CRS string, then:*

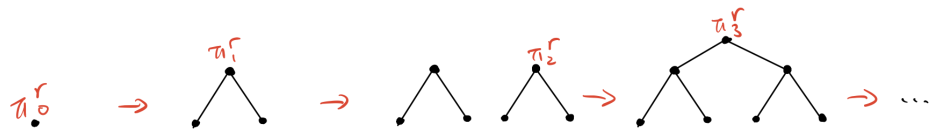
- *(Correctness). The compiled machine respects the original machine. In particular, $m_j$ must be the configuration of $M(\epsilon), \forall j$, independent of $r$ ($\forall r$).*
- *(Completeness). The verifier $V$ accepts the proofs $\pi_j^r$: $\forall r, V(M, j, m_j, \pi_j^r, r) = 1$.*
- *(Computational soundness). $\forall$constant $c$ and $\forall$machine $P'$ such that, $\forall r$ with $|r| = k^2$, $P'$ outputs a triple $(j, m_j'^r, \pi_j'^r)$ in time $K$, we have for sufficiently large $k$ that*

$$\Pr_r[m_j'^r \neq m_j \wedge V(M, j, m_j'^r, \pi_j'^r, r) = 1] < k^{-c}.$$

*$K$ must be super-polynomial for IVC scheme to be secure against poly-time adversaries.*

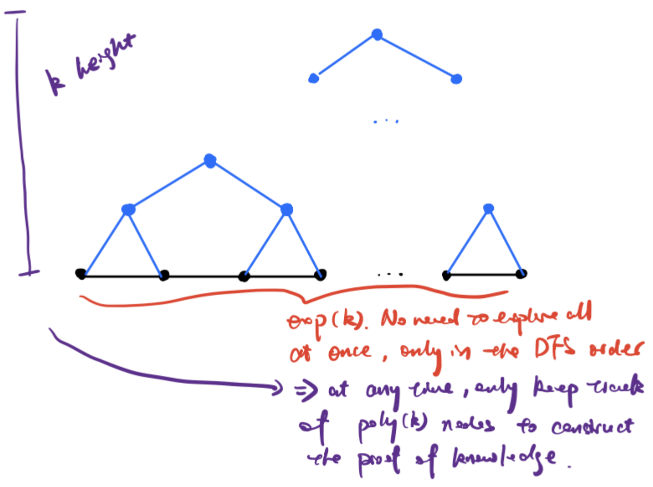
*Proof (main theorem).*

- The construction is a complete binary tree, whose leaves are each of the memory configurations of $M$ in order (say, from left to right). Then, we use the nodes $j$ levels above the leaves as proofs of knowledge of recursive depth $j$ (as such a node covers $2^j$ consecutive configurations).
- Each of these nodes are recursively efficiently constructable from their children, since it only takes one call to $P$ that outputs a single proof for the parent node of two child nodes (later papers instantiated a similar structure using Merkle tree). $P$ by definition 15 takes poly-time.
- Then, we let $C$ be a depth-first traversal of the binary tree, starting at the leaf node that corresponds to time 0, and visit the subsequent leaves in order. Here's an illustration with post-order traversal:



The leaves must be reached from left to right in order, and a **path from root to that leave stored in the stack** is used as a proof of incremental correctness (recall for a DFS, you only need to store nodes from a single root-to-leaf path in the stack at a time at most).

At each node, we use $P$ to get a proof in $poly(k)$ time, and the depth of the recursion is at most the whole tree, which is $k$. So, a leaf is visited every $poly(k)$ time, at which point the stack can be popped to generate the proof for that leaf. So, it takes both $poly(k)$ space and time. Pictorially,


k height
exp(k). No need to explore all at once, only in the DFS order
⇒ at any time, only keep track of poly(k) nodes to construct the proof of knowledge.

- **(The construction in the previous point gives a CS proof).** In the end, the stored proof at the parent node is $(M, s_1, s_2)$, where $s_1$ and $s_2$ are the children of the parent node, each proving their respective time span of their children nodes on the extremes of their respective subtree. Then, $(M, s_1, s_2)$ proves knowledge that $M$ starting from the left extreme of $s_1$ to the right extreme of $s_2$ in $(t_2 - t_1)$ steps. This prover process is sequential in the sense that leaves are proved from left to right, so the proof of knowledge for some leaf on the right must have considered all proofs of knowledge for nodes on the left.
- Given the binary tree from the above construction, $C$:
  - An efficient verifier, $V$, can be constructed to check the correctness of individual proof of the sequence of proofs from the leftmost leaf up to the root (only $k$ of them) and also check if the start and end configurations for each of these proofs match as claimed.
  - By corollary 18, to ensure the probability that proofs accepted are deceptive, we set $\epsilon = k^{-\log k}$; this is because each CS proof as described above takes less than $k^{\log(k)}$ time to construct (as they are $poly(k)$). Then, plug this into corollary 18, we are guaranteed negligible deceptive proofs for running at most $\frac{2k^{-\log k}K'}{(4k^{c_2})^{\log t}} = \frac{2}{4^{\log t}}k^{-\log k - c_2 \log t}K'$.

    How many steps does IVC run? It is executing $C(M)$ and verify for $O(t)$ times, each of the verification takes $k^{O(1)}$. So, the total time is

    $$\underbrace{t \cdot k^{O(1)}}_{\substack{\text{DFS recursion}\\ \text{prover runtime}}} + t \cdot k^{O(1)} \leq O(t \cdot k^{\log(k)}).$$

    Now, $t$ is set to $K$, and we want it to be bounded by the number of steps given according to corollary 18, so

    $$O(K \cdot k^{\log k}) \leq O\left(k^{-\log k - c_2 \log K} K'\right) \implies K k^{2\log k + c_2 \log K} \leq K'$$

    suffices.

  So, this $(C, V, K)$ is a IVC scheme with negligible probability of producing a deceptive proof.

■

## 3.5 Constructing the required noninteractive CS PoK in ROM [Val08]

Finally, a noninteractive CS PoK (see definition 15) can be constructed by replacing the public-coind IP with CRS access model, in definition 15, by queries to a random oracle $\mathcal{R}$ instead. A FS transformation follows.

### 3.5.1 Witness-extractable PCPs

We modify the following aspects of a typical PCP so that it has the **witness-extraction** property:

- A **PCP of proximity (PCPP)** is used in conjunction with an error-correcting code to modify the NP relation.
- The NP witness is padded (or encoded) so that any proof which passes a constant-query check with high probability must be close to the encoding of a valid witness.
- An efficient extractor is then available to decode this proof and recover the NP witness.

These modifications ensure that any prover producing a valid CS proof indeed has seen a corresponding NP witness.

### 3.5.2 CS proof construction via Merkle trees in ROM

This proof need to be short without losing the soundness guarantee (proof merging). It is done using a (Gen, Commit, Open) tuple described as follows, a framework that will be seen again in later constructions.

1. **(PCP proof generation).** The prover computes a witness-extractable PCP proof from the NP witness. This can be done efficiently using the construction in the proof of theorem 19.
2. **(Commitment scheme).** This uses a Merkle tree with strong hash functions instantiated in ROM.
   - The long PCP proof is divided into fixed-size chunks forming the leaves of a binary Merkle hash tree.
   - Internal nodes are computed by applying a hash function to pairs of child nodes.
   - The root of this tree serves as a commitment to the entire PCP proof.
3. **(Challenge scheme).** The root is used as a seed for the random oracle to generate the randomness needed to simulate the PCP verifier's queries.
4. **(Opening scheme).** Instead of sending the full PCP proof, the prover transmits only the responses corresponding to the random queries along with the Merkle verification paths. This significantly reduces the proof length as desired.

### 3.5.3 Verification and knowledge extraction

Finally, we show efficient verification and soudness

- **Verification:** The verifier checks the Merkle paths and recomputes the random queries using the random oracle.
- **Knowledge extraction:** An efficient, straight-line knowledge extractor **simulates the prover and logs all the random oracle queries made during proof generation**. Using these queries, the extractor reconstructs the Merkle tree and recovers the full PCP proof. The extractor then applies the witness extraction algorithm from the witness-extractable PCP to retrieve the NP witness.

It is guaranteed that any prover who manages to produce a valid CS proof efficiently with high probability must indeed possess a valid NP witness.

## 4 Why is uniqueness / unambiguity useful?

It was noted that the construction in section 3 cannot directly be used to show $\mathsf{rSVL}$ hardness because the prover's strategy is not unique. The definition of $\mathsf{rSVL}$ (definition 34) means that there are many false positive solutions, such that, even though one can construct a hard instance on the main computation path using the IVC idea, there could be another proof that also satisfies IVC that is actually easy to solve.

- Uniqueness means that the solution being sought for on the main computation path is the only solution.
- **Incrementally unambiguous computation** helps ensure the solution found is the sink on the main computation line by associating each state with a proof. These proofs have the unambiguous soundness property (unambiguity will be formally defined later, but it means prover must follow a particular proof strategy for the verifier to accept w.h.p.) which guarantees that the verifier outputs 1 on nodes outside of the main computation line with *negligible* probability, making it effectively intractable for any adversary to find such a solution).

In the following two constructions, we summarize two different set of assumptions that lead to IVC constructions that are unambiguous / unique.

## 5 IVC construction #2 in $\mathsf{ROM}$ assuming #SAT-hardness with unambiguity [CHK$^+$19]

This section is reviewing the progress made in [CHK$^+$19]. What [CHK$^+$19] really showed is a reduction from #SAT to $\mathsf{rSVL}$, which is in turn reduced to $\mathsf{EoML}$, which is a problem contained in $\mathsf{CLS}$ (not known to be complete). Note that the first reduction is sufficient by literature today, as it is well-established that $\mathsf{rSVL}$ ($\mathsf{SVL}$ is a promise problem, not total) reduce to $\mathsf{CLS}$.

The high-level intuition of the reduction from #SAT to $\mathsf{rSVL}$ is an IP that unambiguously proved the count of satisfiable assignments of a SAT instance. The unambiguity will be used to imply that, either the false positive solution in the resulting $\mathsf{rSVL}$ is intractable to find or Fiat-Shamir breaks. Then, assuming #SAT is HOA, finding the sink is HOA. See section 5.3.2.

Following that, by showing $\mathsf{rSVL}$ reduces to $\mathsf{EoML} \in \mathsf{CLS}$, it is shown that either #SAT can't be hard which means all the collapses associated with that happen, or #SAT is hard implying $\mathsf{CLS}$ is hard on average.

In summary, the key assumptions are:

1. Fiat-Shamir heuristic is unambiguously sound for the sumcheck protocol relative to $\mathsf{ROM}$. Theorem 28 is an evidence to this end.
2. #SAT is hard. By the nature of this implication, with the assumption in the previous bullet point, if $\mathsf{CLS}$ is not hard-on-average, then #SAT can be efficiently solved.

### 5.1 Interactive $\beta$-prefix sumcheck protocol for $y = \sum_{z_{j+1},\ldots,z_n \in \mathbb{H}} f(\beta_1, \ldots, \beta_j, z_{j+1}, \ldots, z_n)$

The idea is to show that the adaptive soundness of the non-interactive sum-check protocol applied to a particular language $\mathcal{L}_{SC}$ (see definition 25) implies adaptive unambiguous soundness for $\mathcal{L}_{SC}$ (see definition 24).

### 5.1.1 Unambiguous interactive proof protocol

**Definition 22** ($l$-round interactive proof systems). *An interactive proofs protocol is a tuple $(\mathcal{P}, \mathcal{V})$ where $\mathcal{P}$ is the deterministic prover and $\mathcal{V}$ is the probabilistic verifier. It runs for $l$ rounds, and in each $i \in [l]$, $\mathcal{P}$ sends $\alpha_i \in \Sigma^a$ to $\mathcal{V}$, who in turn sends a challenger $\mathcal{B}_i \in \Sigma^b$ back to $\mathcal{P}$. After $l$ such interactions, $\mathcal{V}$ runs on the view*

$$(x, (\beta_1, \ldots, \beta_l), (\alpha_1, \ldots, \alpha_l))$$

*to decide if it should accept. In the case of IP,*

- ***Completeness:** For $x \in \mathcal{L}$, $\mathcal{V}$ accepts with probability $1$ if it interacts with $\mathcal{P}$ on $x$.*
- ***Soundness:** For $x \notin \mathcal{L}$, every cheating $\tilde{\mathcal{P}}$ can only convince $\mathcal{V}$ to accept with probability less than $\delta(|x|)$.*

**Definition 23** (Public-coin IP). *An IP is made public coin if $\beta_i$ is uniformly sampled from $\Sigma^b$. This goes handy with Fiat-Shamir transformation to be made into non-interactive.*

The following definition is slightly modified from the original definition by Reingold-Rothblum-Rothblum in [RRR16], in the sense that the umambiguity is limited to only $x \in \mathcal{L}$. The goal of this definition is to further restrict IP so that the $\mathcal{P}$ should only be a prescribed prover $\mathcal{P}$ that follows a certain proof strategy; if $\mathcal{P}$ fails to do so then $\mathcal{V}$ should also not accept with high probability.

**Definition 24** ($(\delta, \epsilon)$-unambiguously sound interactive proofs protocol). *An IP $(\mathcal{P}, \mathcal{V})$ is $(\epsilon, \delta)$-unambiguously sound IP for $\mathcal{L}$ if*

- ***Prescribed completeness:** For $x \in \{0,1\}^*$, $\mathcal{V}$ outputs $\mathcal{L}(x)$ (i.e. a single bit that indicates if $x \in \mathcal{L}$ or not) with probability $1$ if it interacts with $\mathcal{P}$ on $x$.*
- ***Soundness:** For $x \notin \mathcal{L}$, every (computationally unbounded) cheating $\tilde{\mathcal{P}}$ can only convince $\mathcal{V}$ to accept with probability less than $\delta(|x|)$.*
- ***Unambiguity:** For $x \in \mathcal{L}$, every (computationally unbounded) cheating prover $\tilde{\mathcal{P}}$, every round $i^* \in [l]$ and for every $\mathcal{B}_1, \ldots, \mathcal{B}_{i^*-1}$, if $\tilde{\mathcal{P}}$ first deviates from the protocol in round $i^*$, i.e.*

  $$\tilde{\mathcal{P}}(x, i^*, (\beta_1, \ldots, \beta_{i-1})) \neq \mathcal{P}(x, i^*, (\beta_1, \ldots, \beta_{i-1})),$$

  *then, $\mathcal{V}$ should be run probabilistically over the coin tosses in rounds $i^*, \ldots, l$, and then accepts with probability $\leq \epsilon(|x|)$.*

## 5.2 Explicit construction of an unambiguously sound IP for a particular language

**Definition 25** ($\mathcal{L}_{SC}$). *$\mathcal{L}_{SC}$ is a language defined over tuples of the form*

$$(\mathbb{F}, y, f, \beta_1, \ldots, \beta_j, \tilde{\alpha}_{j+1}, \beta_{j+1}, \ldots, \tilde{\alpha}_{j+i}),$$

*where $i, j \in \{0, \ldots, n\}$ and $i + j \leq n$, such that*

- *If $i = j = 0$, then the tuple is just $(\mathbb{F}, y, f)$, which is the standard input for the sumcheck protocol, which is in $\mathcal{L}_{SC} \iff \sum_{z_1,\ldots,z_n \in \mathbb{F}} f(z_1, \ldots, z_n) = y$.*
- *For $j > 0$ and $i = 0$, then the tuple is $(\mathbb{F}, y, f, \beta_1, \ldots, \beta_j)$, which is in $\mathcal{L}_{SC} \iff$*

  $$\sum_{z_{j+1},\ldots,z_n \in \mathbb{F}} f(\beta_1, \ldots, \beta_j, z_{j+1}, \ldots, z_n) = y.$$

- *For general $j$ and $i \geq 1$, the tuple is of the general form, which is*

$$(\mathbb{F}, y, f, \beta_1, \ldots, \beta_j, \tilde{\alpha}_{j+1}, \beta_{j+2}, \ldots, \tilde{\alpha}_{j+i}).$$

  *It is in $\mathcal{L}_{SC} \iff \tilde{\alpha}_{j+i}$ is consistent with the prescribed prover's final answer, given:*

  - ***the prefixes:** $\beta_1, \ldots, \beta_j$, and*
  - ***the challenges by the verifier:** $\beta_{j+1}, \ldots, \beta_{j+i-1}$.*

**Remark 26.** *Essentially, we are using $\mathcal{L}_{SC}$ as the language to be proved in the sumcheck protocol, and what the sumcheck protocol does is reducing the proof of a tuple being in $\mathcal{L}_{SC}$ to the proof of a longer tuple being in $\mathcal{L}_{SC}$. (The presented protocol is to use $(\mathbb{F}, y, f, \beta_1, \ldots, \beta_n) \in \mathcal{L}_{SC}$ as a proof to show $(\mathbb{F}, y, f, \beta_1, \ldots, \beta_j) \in \mathcal{L}_{SC}, \forall j < n$).*

**Claim 27.** *If the sumcheck protocol is an **adaptively sound argument system** for the language $\mathcal{L}_{SC}$, then it is also an **adaptively unambiguously sound argument system** for $\mathcal{L}_{SC}$*

**Theorem 28.** *Let $f : \mathbb{F}^n \to \mathbb{F}$ be an $n$-variate polynomial of degree at most $d < |\mathbb{F}|$ in each variable. We have a sumcheck protocol that is a $\left(\frac{d(n-j)}{|\mathbb{F}|}\right)$-unambiguously sound IP for prefixed $\mathcal{L}_{SC}$, i.e. consider definition 24 with the following changes:*

- *$\mathcal{L}$ = prefixed-$\mathcal{L}_{SC}$, which means we use the second case of definition 25, and we check*

$$\sum_{z_{j+1},\ldots,z_n \in \mathbb{F}} f(\beta_1, \ldots, \beta_j, z_{j+1}, \ldots, z_n) = y.$$

  *In particular, it is to decide if the given $(f, y, \vec{\beta})$ belongs to $\mathcal{L}_{SC}(f, y, \vec{\beta}, \emptyset)$*

- *Relax the checking to $\mathbb{H} \subseteq \mathbb{F}$, i.e.*

$$\sum_{z_{j+1},\ldots,z_n \in \mathbb{H}} f(\beta_1, \ldots, \beta_j, z_{j+1}, \ldots, z_n) = y.$$

  *So, a tuple is a member of $\mathcal{L}_{SC}$ if the above inequality holds, not a member otherwise.*

- *Let $\delta(n) = \frac{d(n-j)}{|\mathbb{F}|}$.*

- *Let $\epsilon(n) = \frac{d(n-j)}{|\mathbb{F}|}$. Since $\delta = \epsilon$, we simplified the notation to just $\left(\frac{d(n-j)}{|\mathbb{F}|}\right)$-unambiguously sound IP as stated in the theorem.*

*Proof.* We first define sumcheck protocol in the context of the specific language, $\mathcal{L}_{SC}$, we have here.

**Definition 29** (Sumcheck protocol (for $\mathcal{L}_{SC}$))**.** *Recall $\mathbb{H} \subseteq \mathbb{F}$ and $\vec{\beta} = (\beta_1, \ldots, \beta_j)$ is prefix fixed up to the current round of interaction, then a round of an prefixed sumcheck protocol interactions is the following: iterate through $i \in [j+1, n]$*

1. *$\mathcal{P}_{SC}$, the prover of $\mathcal{L}_{SC}$ membership, first computes*

$$g_i(x) := \sum_{z_{i+1},\ldots,z_n \in \mathbb{H}} f(\beta_1, \ldots, \beta_{i-1}, x, z_{i+1}, \ldots, z_n).$$

   *Then, $\mathcal{P}_{SC}$ can evaluate $g_i(x)$ at any $(d+1)$ points to commit to $\mathcal{V}_{SC}$ this computed polynomial. WLOG, let $g_i$ compute the first $(d+1)$ elements of $\mathbb{F}$, and let the values be*

$$\{g_i(\gamma)\}_{\gamma=0}^{d}.$$

2. *Upon the reception of these $(d+1)$ values at the first $(d+1)$ points of the field, $\mathcal{V}_{SC}$ interpolates the degree-d polynomial uniquely; let this interpolated polynomial be $\tilde{g}_i$. Then, it does two things:*

   (a) *$\mathcal{V}_{SC}$ holds the current sum based on the previous round's challenge (will show how that is computed for the current round next), so it first checks if the current interpolated polynomial agrees with that expected value:*
   $$y_{i-1} \stackrel{?}{=} \sum_{x \in \mathbb{H}} \tilde{g}_i(x).$$
   *If not, $\mathcal{V}_{SC}$ rejects here.*

   (b) *If $\mathcal{V}_{SC}$ didn't reject from the inconsistency, then it now randomly samples a new challenge $\beta_i \in_R \mathbb{F}$ **(the fact that this challenge is uniformly randomly sampled is important for this protocol to be made non-interactive in** ROM **using the Fiat-Shamir heuristic)**, and sets*
   $$y_i = \tilde{g}_i(\beta_i),$$
   *and sends $\beta_i$ to $\mathcal{P}_{SC}$.*

*By the end, $\mathcal{V}_{SC}$ checks if $y_n \stackrel{?}{=} f(\beta_1, \dots, \beta_n)$. $\mathcal{V}_{SC}$ accepts if the equality holds; rejects otherwise.*

Now, we show that this described protocol is a $\left(\frac{d(n-j)}{|\mathbb{F}|}\right)$-unambiguously sound IP for prefixed $\mathcal{L}_{SC}$:

- **Prescribed completeness:** By the construction, given $y$ and $f$, we compute series of $g_i$ from $i = j+1$ up to $n$, and accept only if the final $y_n$ is as expected, which happens only if the original sum is $y$. So, the final decision of membership is guaranteed to be correct if the sumcheck protocol is followed exactly.

- $\frac{d(n-j)}{|\mathbb{F}|}$**-soundness:** Let $f : \mathbb{F}^n \to \mathbb{F}$ be an arbitrary polynomial with degree $\leq d$ in each variable such that
  $$\sum_{z_1, \dots, z_n \in \mathbb{H}} f(z_1, \dots, z_n) \neq y.$$
  Suppose to the contrary that there is such a $\tilde{\mathcal{P}}$ that can make the sumcheck protocol accept with probability greater than
  $$\frac{dn}{|\mathbb{F}|}.$$
  Let

  - $S$ be the event that $(\tilde{\mathcal{P}}(f, y, \emptyset), \mathcal{V}(f, y, \emptyset))$ is accepted by the sumcheck protocol,.
  - $A_i$ be the event that
  $$g_i(x) = \sum_{z_{i+1}, \dots, z_n \in \mathbb{H}} f(\beta_1, \dots, \beta_{i-1}, x, z_{i+1}, \dots, z_n).$$

  We first note that $\Pr[S \mid A_1 \wedge \ldots \wedge A_n] = 0$, because, with the conditional, the sumcheck protocol must have been followed entirely (maybe prover can do additional things, but the $\mathcal{V}$ already acquires all the necessary information for it to decide, and it can just ignore the additional information from $\mathcal{P}$), by prescribed completeness $\mathcal{V}_{SC}$ must eventually reject.

  We draw the contradiction using the assumption that $\Pr[S] \geq \frac{dn}{|F|}$, by inductively adding in the conditional and getting
  $$\Pr[S \mid A_1 \wedge \cdots \wedge A_n] \geq s - \frac{dn}{|\mathbb{F}|} > 0.$$
  :

1. **(Base case).** We show that
$$\Pr[S \mid A_n] \geq s - \frac{d}{|\mathbb{F}|}$$
Here's how: we first have
$$s = \Pr[S] = \Pr[S \mid A_n] + \Pr[S \mid \neg A_n],$$
where, by the Schwartz-Zippel lemma, which states that two distinct polynomials with degree $\geq d$ over $\mathbb{F}$ can meet in at most $d$ points,
$$\Pr[S \mid \neg A_n] \leq \frac{d}{|\mathbb{F}|}.$$
In more details, this is because, in the event of "$\neg A_n$",
$$g_n(x) \neq f(\beta_1, \ldots, \beta_{n-1}, x),$$
which means the interpolated function is a different function from $f$. To have the protocol accept,
$$\Pr[S \mid \neg A_n] = \Pr[\sum_{x \in \mathbb{H}} g_n(x) = y_{n-1} \wedge y_n = g_n(\beta_n)] \leq \Pr[y_n = g_n(\beta_n)],$$
which is the probability that the final random challenge $\mathcal{V}_{SC}$ selects turns out to be where $g_n$ and $f$ meet, which happens at at most $d$ points, so
$$\Pr[S \mid \neg A_n] \leq \Pr[y_n = g_n(\beta_n)] \leq \frac{n}{|\mathbb{F}|}.$$
So,
$$s = \Pr[S] = \Pr[S \mid A_n] + \Pr[S \mid \neg A_n] \leq \Pr[S \mid A_n] + \frac{n}{|\mathbb{F}|} \implies \Pr[S \mid A_n] \geq s - \frac{n}{|\mathbb{F}|}.$$

2. **(Induction).** This is for the exact same reason but for the selection of a different value. In particular:
$$\Pr[S \mid \neg(A_{j-1}) \wedge A_j \wedge \ldots \wedge A_n] \leq \Pr[\beta_{j-1} \in \{\text{the } \leq d \text{ points where distinct } g_{j-1} \text{ and } f \text{ meet}\}] \leq \frac{d}{|\mathbb{F}|}.$$
So, with the inductive assumption that
$$\Pr[S \mid A_j \wedge \ldots \wedge A_n] = \Pr[S \mid \neg(A_{j-1}) \wedge A_j \wedge \ldots \wedge A_n] + \Pr[S \mid A_{j-1} \wedge A_j \wedge \ldots \wedge A_n] \geq s - \frac{(n-j+1)d}{|\mathbb{F}|}$$
$$\implies \Pr[S \mid A_{j-1} \wedge A_j \wedge \ldots \wedge A_n] \geq s - \frac{(n-j+1)d}{|\mathbb{F}|} - \Pr[S \mid \neg(A_{j-1}) \wedge A_j \wedge \ldots \wedge A_n] \geq s - \frac{(n-j+1)d}{|\mathbb{F}|} - \frac{d}{|\mathbb{F}|},$$
which indeed gives us
$$\Pr[S \mid A_{j-1} \wedge A_j \wedge \ldots \wedge A_n] \geq s - \frac{(n-(j-1)+1)d}{|\mathbb{F}|}.$$

Thus, by the conditional reaching $j$, we have
$$\Pr[S \mid A_j \wedge \cdots \wedge A_n] \geq s - \frac{(n-1+1)d}{|\mathbb{F}|} = s - \frac{nd}{|\mathbb{F}|} > \underbrace{\frac{nd}{|F|}}_{\text{assumption}} - \frac{nd}{|F|} = 0,$$
so $s \leq \frac{nd}{|F|}$, proving the soundness with this parameter. (Proof follows similarly, where instead of $j = n$, it is some $j$ in the middle, to give us $s \leq \frac{(n-j)d}{|\mathbb{F}|}$.)

- **($\frac{(n-j)d}{|\mathbb{F}|}$-unambiguity).** Suppose $f$ is correctly defined (i.e. the correct case, with $d$, $\mathbb{F}$, $y$, all defined regularly, with
$$\sum_{z_1,\ldots,z_n\in\mathbb{H}} f(z_1,\ldots,z_n)=y).$$
Further, let $i^*$ be the first time step where $\tilde{\mathcal{P}}$ does something different from the honest $\mathcal{P}$, so
$$g_i(x)\equiv\tilde{g}_i(x),\forall i<i^*; g_{i^*}(x)\not\equiv\tilde{g}_{i^*}(x).$$
For the rounds of interactions after $i^*$, we have the following claim:
**Claim 30.** *For $i\in[i^*,n-1]$ and $\beta_{i^*},\ldots,\beta_{i-1}\in\mathbb{F}$ and suppose $g_i\not\equiv\tilde{g}_i$. Then,*
$$\Pr_{\beta_i\in_R\mathbb{H}}[\mathcal{V}_{SC}\text{ rejects}\vee g_{i+1}\not\equiv\tilde{g}_{i+1}]\geq 1-\frac{d}{|\mathbb{F}|}.$$
*Proof (claim).* By Schwartz-Zippel lemma again, with probability at least $(1-\frac{d}{|\mathbb{F}|})$ the uniformly chosen $\beta_i$ leads to
$$g_i(\beta_i)\neq\tilde{g}_i(\beta_i).$$
In this, if further $g_{i+1}=\tilde{g}_{i+1}$, then the following must be true
$$\sum_{x\in\mathbb{H}}\tilde{g}_{i+1}(x)=\sum_{x\in\mathbb{H}}g_{i+1}(x)=g_i(\beta_i)\neq\tilde{g}_i(\beta_i)=y_i,$$
so $\mathcal{V}_{SC}$ will necessarily reject when checking
$$\sum_{x\in\mathbb{H}}\tilde{g}_{i+1}(x)=y_i.$$
■

In other words, the above fails to reject with probability $\leq\frac{d}{|\mathbb{F}|}$, which means, overall $i\in[i^*,n-1]$, the probability of rejecting somewhere in between these steps is, by a simple union bound on the above failure probability,
$$\geq 1-\frac{|[i^*,n-1]|d}{|\mathbb{F}|}\geq 1-\frac{(n-1)d}{|\mathbb{F}|}.$$
Furthermore, applying Schwartz-Zippel again on the last case, which is where $g_n\not\equiv\tilde{g}_n$ but the choice of $\beta_n$ somehow is one of the $\leq d$ points where
$$\Pr[g_n(\beta_n)=\tilde{g}_n(\beta_n)]\leq\frac{d}{|\mathbb{F}|},$$
so, also union bounding the above failure probability with this failure probability at the last step, we have
$$\Pr[\mathcal{V}_{SC}\text{ accepts, incorrectly}]\leq\frac{nd}{|\mathbb{F}|}.$$
(Proof follows similarly, for up to $j$, instead of $n$, then this factor would become $\leq\frac{(n-j)d}{|\mathbb{F}|}$ as the above proof was taking over the union bound anyways.) ■

**Remark 31.** *Note that if $j=0$, then just run the sumcheck protocol in definition 29 in its fullness, i.e. $i$ should loop from 1 to $n$. If $j>0$, then it is just run partially, with prefix $\vec{\beta}$ already specified and $\vec{\tau}$ is the transcription already done so far starting from $i=j+1$, i.e.*
$$\vec{\beta}=(\beta_1,\ldots,\beta_j);\vec{\tau}=(\tilde{\alpha}_{j+1},\beta_{j+1},\ldots,\tilde{\alpha}_{j+i}).$$
*In this case, the protocol is*
$$(\mathcal{P}_{SC}(y,f,\vec{\beta},\vec{\tau}),\mathcal{V}_{SC}(y,f,\vec{\beta},\vec{\tau})).$$
*This protocol analogously follows from the proof for theorem 28 that it is $\frac{d(n-i-j)}{|\mathbb{F}|}$-unambiguously sound IP system for $\mathcal{L}_{SC}$.*

## 5.3 #SAT $\leq$ rSVL

### 5.3.1 Arithmetization of SAT

To use theorem 28 on solving #SAT, we first review how to translate a SAT formula into a low-degree polynomial.

- SAT $\leq$ 3SAT using the standard trick of splitting classes by introducing new variables. Since the size of the largest clause is constant, the number of new variables and clauses introduced will only lead to a constant factor blow-up of the original formula.

- 3SAT $\leq$ 3SAT-4, where 3SAT-4 means each variable appears in at most 4 clauses (this corresponds to the degree of the final polynomial, so it is colloquially called a degree of the SAT formula too). Suppose some variable $x_i$ is in more than 4 clauses. Then, create a new variable $x_i'$ and then associate the two as $(\neg x_i \vee x_i') \wedge (x_i \vee \neg x_i')$, which is the same as $x_i \iff x_i'$, and then distribute the original occurrences of $x_i$ into $x_i$ and the new $x_i'$ as evenly as possible. So, the occurrence of each is at most $\lceil \frac{n}{2} \rceil + 2 \leq n$ for $n > 4$ (the edge case of 5, we create the "$\iff$" relationship between three variables at once).

- 3SAT-4 $\leq$ a low-degree polynomial. This is done by arithmetization, using the following translations:

  - **(AND).** $a \wedge b \mapsto a \cdot b$.
  - **(NOT).** $\neg a \mapsto (1 - b)$
  - **(OR).** $a \vee b \mapsto 1 - (1 - a)(1 - b)$.

  Clearly, if any variable occurs at most 4 times, the degree on each variable is at most 4. Thus, the final polynomial has a low degree.

### 5.3.2 The reduction (set-up and statement)

Firstly, let's start with a Boolean formula. By section 5.3.1, this Boolean formula can WLOG be thought of as the a $n$-variate polynomial function of degree $d$ (low degree). The goal is to construct a hard rSVL assuming that counting the number of satisfiable assignments of such a Boolean formula is hard on average, and the approach is using sumcheck protocol that is made non-interactive by using Fiat-Shamir heuristic that is considered secure in ROM (section 2.2.4).

Let's then recall the definition of an rSVL (which stands for the relaxed SVL). Before that, we first define the regular SVL.

**Definition 32** (Sink-of-Verifiable-Line (SVL))**.** *An* SVL *problem is a* 4*-tuple* $(S, V, T, v_0)$*, each of which are defined as follows:*

- $v_0$ *is the given starting point of the computation; it can also be thought of as a special source node of a directed graph.*

- $T$ *marks the total number of time steps. For a non-trivial* SVL *instance (as is typical for* TFNP*),* $T$ *is superpolynomial, and usually* $2^n$*. Here, we let it generally be* $T \in [2^n]$*.*

- $S : \{0,1\}^n \to \{0,1\}^n$*, which is the successor circuit. That is, it takes the label of a node as input, which is a* $n$*-bitstring, and then output the next node.*

- $V : \{0,1\}^n \times [T] \to \{0,1\}$*, which is the verification circuit. What it does is the following: it takes a node and a number that represents the current time step as the inputs. It is* ***promised*** *that*

$$V(v, i) = 1 \iff S^i(v_0) = v.$$

*The goal is to find a $v \in \{0,1\}^n$ such that $V(v,T) = 1$ (i.e. a $v$ such that $S^T(v_0) = v$).*

**Remark 33.** SVL *would be an instance of a promise problem, since not all possible $V$ is honestly implemented. We make* SVL *a total problem by introducing another rejection condition in* rSVL*.*

**Definition 34** (Relaxed SVL (rSVL))**.** rSVL *has exactly the same set-up as* SVL*, except that, in addition to accepting a $v$ where $V(v,T) = 1$ as a solution, it also accepts a solution for the false positive (i.e. to testify that $V$ is incorrectly implemented). Thus, either of the following is a solution:*

- ***Sink:*** *$v \in \{0,1\}^n$ such that $V(v,T) = 1$.*
- ***False positive:*** *A pair $(v,i) \in \{0,1\} \times [2^n]$ such that $V(v,i) = 1$ but $S^i(v_0) \neq v$.*

With the arbitrary degree-$d$ $n$-variate polynomials we started with, we build the following instance of rSVL based on an unambiguously sound non-interactive sumcheck protocol for proving the count of #SAT. The main line of computation of the rSVL instance is a line that:

- Starts at $v_0 = (0^n, f(0^n), \emptyset)$.
- Let $T = 2^n$, so the line of computation ends at $(2^n, y_{2^n}, \pi_{2^n})$.
- Successor circuit $S$: see section 5.3.3.
- Verification circuit $V$: see section 5.3.4.

### 5.3.3 The successor circuit, $S = S_n$

This construction is leveraging the properties of a particular sumcheck protocol construction proved in the section that culminates to theorem 28, then this suncheck protocol is made non-interactive using Fiat-Shamir transformation as summarized in section 2.2.4 (see remark 10).

The construction is recursive, and as such we describe what the base case $S_0$ does and how $S_{n-j+1}$ is constructed form $S_{n-j}$. Ultimately, $S_n$ is constructed to represent the successor circuit. At each level of the construction, it takes the following inputs:

$$\left(\vec{\beta}, s, \vec{t}, \mathsf{tran}\right),$$

where

- $\vec{\beta} = (\beta_1, \ldots, \beta_{j-1}) \in \mathbb{F}^{j-1}$ is prefix for the sumcheck protocol. In particular, out of the $n$ variables of the Boolean formula we started with, the first $(j-1)$ variables are fixed.
- $s_i \in \mathbb{F}^*$ is a state, that is a set of pairs of prefix sums and their proofs, i.e. $s_i = \{(y_1, \pi_1), (y_2, \pi_2), \ldots\}$. notice that this list of proofs and partial sum $y_i$ only goes up to $n$, so the size is polynomial (we can write the sum $y \in [0, 2^n]$ in $n$ bits in binary).
- $t = (t_1, \ldots, t_l) \in [d+1]^{\leq n-j+1}$ is an index. In short, it represents the location of a node on the $(d+2)$-ary graph from unwinding the recursion construction (will show why this "$d+2$" is the case in the induction below), if all the nodes are ordered by when they are last visited by the $DFS$ traversal.
- $\mathsf{tran}$: the transcription of the execution of the sumcheck protocol so far.

With these input, we describe what each level of the recursion does from the output of the previous level:

- **(Base case).** $S_0$ is the base case. As such, it has nothing to trace downward using $t$, and it should not have any transcript (as the sumcheck protocol should just be started), which means it takes $(\vec{\beta}, s, \emptyset, \emptyset)$ as the input. This should only be accepted $\iff$

$$y = f(\vec{\beta}).$$

- **(Induction).** Given $S_{n-j}$, we construct $S_{n-j+1}$. $S_{n-j+1}$ can be constructed by making $(d+2)$ queries to $S_{n-j}$, where
  1. The first $(d+1)$ queries compute the values of $g_{n-j}$ on $(d+1)$ values, and as $g_{n-j}$ would have degree at most $d$, these points are enough to interpolate the polynomial.
  2. Then, with another query to $S_{n-j}$, we compute the value for the challenge input and check if
  $$y_{n-j} \overset{?}{=} g_{n-j}(\beta_{n-j}).$$
  If this equality holds, then append a proof and this partial sum to the existing proof, which makes the proof
  $$((\pi_1, y_1), \ldots, (\pi_{n-j}, y_{n-j})),$$
  now.

  **The way this is defined recursively on $S_{n-j}\left(\vec{\beta}, s, \vec{t}, \mathsf{tran}\right)$ for $S_{n-j+1}$ is the following in particular:**

  – If $t = \epsilon$, that means the index is pointing at the root, which means it is at the final state. If $s$, which records the proof, is set, then return $s$ (this is the sink of the circuit described, as we are making $s$ the successor of itself).

  – If $t = d+1$, then sufficient information is acquired to compute $\{g_{n-j}(\gamma)\}_{\gamma=0}^{d}$, so merge these proofs and the next state is the final state of $(S_{n-j}, V_{n-j})$. So, return $\{g_{n-j}(\gamma)\}_{\gamma=0}^{d}$, appended to $\pi_{d+1}$.

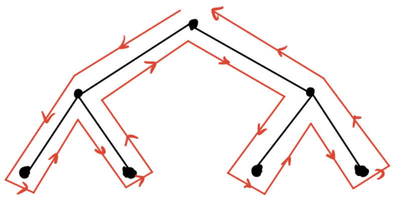
  – The rest is for the case $t \leq d$. Then, we look at the sub-state execution, starting from the substate pointed to by $t_1$. So we pay attention to the proofs truncated up to $t_1$ by computing the sub-state
  $$s' = \{(y_\gamma, \pi_\gamma)\}_{\gamma}^{t_1-1},$$
  and move on to the index stemming from $s'$, which is just
  $$\vec{t'} = (t_2, \ldots, t_l).$$
  Intuitively, this is restarting the computation from the last, $t_1$-th, sub-state of the current state.

    * If $t = d$ or $t_1 = d+1$ (i.e. the last execution before the final state execution):
      1. $\mathsf{tran}$ should be initialized if it has not been. It should be the sum of currently interpolated polynomial evaluated at 0 and 1 and with the prefix:
      $$\mathsf{tran} = (\mathbb{F}, y_0 + y_1, f, \vec{\beta}).$$
      2. If $\mathsf{tran}$ exists, then we append $\{y_\gamma\}_{\gamma=0}^{t_1-1}$ to it.
      3. Finally, we prepare the challenge for the final proof merging step (since the next step would reach $(d+2)$-th execution), where the challenge input is computed non-interactively
      $$\sigma = h(\mathsf{tran}).$$
      This $\sigma$ should also be appended to $\mathsf{tran}$.
      4. Finally, initialize $(d+2)$-th sub-state for the next round which is to compute the final sub-state. The next sub-state should be
      $$s' = S_{n-j}((\vec{\beta}, \sigma), \vec{t'}, s', \mathsf{tran}),$$
      where the prefix vector is appended by the new challenge $\sigma$. Append $\{(y_\gamma, \pi_\gamma)\}_{\gamma=0}^{d}$ to $s'$ which in turn updates $s$, and so return.

$*$ If neither $t = d$ nor $t_1 = d + 1$, so we are not yet one step before the last execution of the $(d + 1)$-th sub-state of the current state, so we should continue the computation (below just the sub-state of the current state):

1. If $t_1$-th state is final, then we initialize $(t_1 + 1)$-th sub-state. by appending $\{(y_\gamma, \pi_\gamma)\}_{\gamma=0}^{t_1}$ to $S_{n-j}((\vec{\beta}, t_1 + 1), \epsilon, \emptyset, \emptyset)$.
2. If $t_1$-th state is non-final, then we continue the $t_1$-th sub-state. by appending $\{(y_\gamma, \pi_\gamma)\}_{\gamma=0}^{t_1-1}$ to $S_{n-j}((\vec{\beta}, t_1), \vec{t'}, s', \emptyset)$.

**What $S = S_n$ does?** It keeps track of the recursive computation in the DFS order of the unwinded recursion tree, a perfect $(d + 2)$-ary tree. Every application of $S$ is to further this computation by one step, which is equivalent to moving one step according to the *DFS* traversal. Intuition: the order of computation as unpacked above should be in the following order of the unwound recursion tree:



Concretely, the main computation line described by $S$ is

$$(0^n, s_1), (0^{n-1}1, s_2), (0^{n-1}, s_3), (0^{n-2}10, s_4), (0^{n-1}11, s_5), \ldots, (1, S_{T-1}), (\epsilon, S_T),$$

where each of these nodes are $(\vec{t}, s_i)$, where $s_i$ is the $i$-th vertex on the recursion tree by the DFS traversal order that $\vec{t}$ should point to.

**Remark 35.** ***Efficient runtime:*** *Note that $S_0$ consists of trivial computations, so $|S_0| = poly(n)$. Then,*

$$|S_{n-j+1}| < |S_{n-j}| + poly(n),$$

*as $S_{n-j+1}$ sequentially calls $S_{n-j}$ for $(d + 2)$ many times, plus efficient operations like appending etc. Thus, the final*

$$|S_n| = poly(n),$$

*which means the circuit is efficiently computable.*

***Memory and proof size:*** *The following can be found by solving the recursion and realizing that $M(0) = P(0) = \log(p)$ where $p = |\mathbb{F}|$:*

- *Memory size bound: $M(n) \leq (d + 1)n \log(p)$.*
- *Proof size bound: $P(n) \leq (d + 1)n \log(p)$.*

### 5.3.4 The verification circuit, $V = V_n$

Given a node $(t, s_i)$, what the verification circuit does, intuitively, is simply checking if the $i$-th index of the unwound recursion tree in the DFS order is indeed the vertex pointed to by $\vec{t}$. To catch this intuition, the recursive construction of $V = V_n$ based on $V_i$ all the way down to $i = 0$ is reciprocal of the construction for $S$. Here is a sketch of how this verification circuit is recursively built:

- Inputs: $\vec{\beta} \in \mathbb{F}^{j-1}$, $\vec{t} \in [0, d + 1]^{\leq n-j+1}$, $s$ as a set of pairs of prefix sums and proofs, and $\mathsf{tran}$. These are exactly the same as that of constructing $S_{n-j+1}$.

- If we reach the base case where $V_0(\vec{\beta}, \epsilon, \{(y, \emptyset)\}, \emptyset)$, then the verifier should accept $\iff$ $y = f(\vec{\beta})$. (Note that the reason why the proof for this $y$ here is just $\emptyset$ because a direct computation can be done efficiently.)
- If it's not the base case, we should be able to construct $V_{n-j+1}$ efficiently from $V_{n-j}(\vec{\beta}, \vec{t}, s, \mathsf{tran})$:
  1. If $t = \epsilon$, then we are at the root, meaning that it is the final state of $V_{n-j+1}$, which means we should return
  $$b \leftarrow \mathcal{V}_{SF}((y_\epsilon, f, \vec{\beta}), \pi_\epsilon).$$
  2. If not, then we are in an intermediate state, in which case we need to make queries to $V_{n-j}$ to verify all final sub-states of the root, i.e. checking
  $$V_{n-j}((\vec{\beta}, \gamma), \epsilon(y_\gamma, \pi_\gamma), \emptyset),$$
  for all $\gamma \in [0, t_1 - 1]$. Reject if any of these queries reject, and $t_1$ is bounded by $d + 1$.
     - Note that in this case, sub-states are only final to the indexed $t_1$, which is why we verify computations up to $(t_1 - 1)$.
     - For the $t_1$-th sub-state, we need to decide what that state is, and it should be the first $t_1$ partial sums and proofs from $s$, i.e.
     $$s' = \{(y_\gamma, \pi_\gamma)\}_{\gamma=0}^{t_1-1}.$$
     Similarly, $t'$ should start at after $t_1$, i.e.
     $$t' = (t_2, \ldots, t_l).$$
  3. Now, if $t_1 = d + 1$, i.e. right before the final state, then we should do the following:
     (a) Make sure tran is initialized and is
     $$\mathsf{tran} = (\mathbb{F}, y_0 + y_1, f, \beta).$$
     (b) Update this transcript by appending
     $$\{y_\gamma\}_{\gamma=0}^{d}.$$
     (c) This is the special step, which is including the challenge point as an extra part of the proof (again, as part of the sumcheck protocol). This is done by
     $$\sigma = h(\mathsf{tran}),$$
     and append $\sigma$ to tran.
     (d) Make a final, $(d + 2)$-th query to the $V_{n-j}$ circuit, which is to confirm the challenge value is as expected. If $V_{n-j}$ rejects, $V_{n-j+1}$ rejects.
  4. Finally, if $t_1 < d+1$, no need for transcript as there is no need to come up with the challenge yet, so we simply verify if the sub-state information is valid by sumcheck protocol, i.e. checking
  $$V_{n-j}((\vec{\beta}, t_1), \vec{t'}, s', \emptyset).$$
  Rject if $V_{n-j}$ rejects.

  If no rejection happens by the end, $V_{n-j+1}$ should accept.

Then, $V = V_n$. For the size of the verification circuit and its efficiency for the sake of a valid rSVL instance, the argument is analogous to that in remark 35.

### 5.3.5 Hardness of this rSVL instance

To finish off this proof, we still need to show the above constructed instance satisfies the following properties to make it a hard rSVL instance:

**Theorem 36.** *For security parameter $n$, fix a finite field $\mathbb{F}$ of sufficiently large $p$ (e.g. $O(2^n)$). Let $f$ be an $n$-variate polynomial over $\mathbb{F}$ of individual degree $\leq d$. Pick a hash function $h$ uniformly at random from a family of cryptographic hash functions $\mathcal{H}$. Let*

$$S := S_{f,h} : \{0,1\}^M \to \{0,1\}^M \text{ and } V := V_{f,h} : \{0,1\}^M \times [T] \to \{0,1\}^M.$$

*(Note this is the same construction as above $(S_n, V_n)$. We just encode in the circuit what $h$ we use to sample challenges and what polynomial $f$ we are proving the sumcheck protocol on. As such, the construction and the polynomial computability of $S_{f,h}$ and $V_{f,h}$ follow from above). Let $\mathcal{A}$ be an arbitrary adversary that solves instances from*

$$\{(S, V, (0^n, f(0^n), \emptyset), T)\}_{n\in\mathbb{N}}$$

*in polynomial $T_{\mathcal{A}}$. Let $\mathcal{N}(n)$ be non-negligible. $\mathcal{A}$ must do one of the following:*

- ***(Finding a solution on the main computation line is hard)*** *Find the correct $\sum_{z_1,\ldots,z_n\in\{0,1\}} f(z_1,\ldots,z_n)$ in time $O(T_{\mathcal{A}})$ w.p. $\geq \mathcal{N}$.*
- ***(Finding a solution off the main computation path is intractable)*** *Break unambiguous soundness of the non-interactive sumcheck protocol with probability $\geq \frac{\mathcal{N}}{(d+1)\cdot n}$*

*Proof.* The proof is simply realizing that each of these two cases correspond to the two types of solutions listed in definition 34. ■

**Theorem 37.** *In* ROM*, #SAT $\leq$ rSVL.*

*Proof.* In ROM, we can assume that Fiat-Shamir heuristic is unambiguously sound for the sumcheck protocol from theorem 28. This is because we have a family of ideal hash functions $\mathcal{H}$ that the sumcheck protocol from theorem 28 can be made unambiguously sound as claimed for $\mathcal{L}_{SC}$, as was shown.

We use this conclusion to show that the second type of solution as described in theorem 36 is impossible. Suppose $f$ is a low-degree polynomial with an arbitrary SAT formula underlying it (it is WLOG a 3SAT-4 formula), and then we run the sumcheck protocol on it. Suppose to the contrary that $v$ is the false positive solution found, and it is

$$v = (t, \{(\tilde{y}_1, \tilde{\pi}_1), \ldots, (\tilde{y}_l, \tilde{\pi}_l)\}); V(v, i) = 1, S^i((0^n, f(0^n), \emptyset)) \neq v.$$

Suppose the following be the real solution that happens after applying $S$ on the starting state for $i$ times:

$$v_i := (t, \{(y_1, \pi_1), \ldots, (y_l, \pi_l)\}).$$

- **(Either soundness or unambiguity must be violated).** In the case that neither the soundness nor the unambiguity were broken, we show that it must be the case that $v_i = v$, by inducting on the following equality: $(\tilde{y}_i, \tilde{\pi}_i) = (y_i, \pi_i)$.
  - **(Base case).** The base case is trivial, as only the same summing procedure happened in both cases, as we assume soundness and unambiguity/
  - **(Induction).** Now suppose that $(\tilde{y}_i, \tilde{\pi}_i) = (y_i, \pi_i)$ for $i \in [k-1]$. Firstly $\tilde{\beta}_k = \beta_k$ must be correctly computed, as, by the inductive hypothesis, the hash of the transcript up to this point must be the same (violating this would be caught by the verifier). Then, since soundness and unambiguity are assumed, it must be the case that

$$(\tilde{y}_k, \tilde{\pi}_k) = (y_k, \pi_k).$$

Thus, it cannot be the case that $S^i((0^n, f(0^n), \emptyset)) \neq v$ if both soundness and unambiguity are preserved.

- **(Consequences of lack of either soundness or unambiguity).** As a result of the violation as described, $\exists k' \in [1, l]$ such that at least one of the following things would happen:
  - (Incorrect solution is accepted).
$$\mathcal{V}_{FS}((\tilde{y}_{k'}, f, \tilde{\vec{\beta}}_{k'}), \tilde{\pi}_{k'}) = 1, \text{ but } \sum_{z_{j_{k'}+1}, \ldots, z_n \in \{0,1\}} f(\tilde{\vec{\beta}}_{k'}, z_{j_{k'}+1}, \ldots, z_n) \neq \tilde{y}_{k'}.$$
  - (Prescribed proof strategy is not followed).
$$\mathcal{V}_{FS}((\tilde{y}_{k'}, f, \tilde{\vec{\beta}}_{k'}), \tilde{\pi}_{k'}) = 1, \text{ but } \tilde{\pi}_{k'} \neq \mathcal{P}_{FS}(\tilde{y}_{k'}, f, \tilde{\vec{\beta}}_{k'}).$$

  Thus, we construct $\mathcal{A}'$ that breaks either the soundness or the unambiguity of the underlying Fiat-Shamir heuristic on the sumcheck protocol for $\mathcal{L}_{SC}$ (consequently contradicting theorem 28):

  1. In the process of running, $\mathcal{A}'$ randomly samples $k' \leftarrow [1, l]$.
  2. Return, instead, the following
$$((f, \tilde{\vec{\beta}}_{k'}, \tilde{y}_{k'}), \tilde{\pi}_{k'}),$$
     so that teh rest of the protocol follows what $\mathcal{A}$ would do.

  If $k'$ is the correct guess by $\mathcal{A}'$, then $\mathcal{A}'$ can just follow the rest of $\mathcal{A}$ to succeed (and there may or may not be other cases where $\mathcal{A}'$ succeeds without guessing $k'$ correctly):
$$\begin{aligned}\Pr[\mathcal{A}' \text{ succeeds}] &\geq \underbrace{\Pr[\mathcal{A} \text{ succeeds}]}_{\geq \mathcal{N}} \cdot \underbrace{\Pr[k' \text{ is a correct guess} \mid \mathcal{A}' \text{ succeeds}]}_{\geq \frac{1}{l}} \\ &\geq \frac{\mathcal{N}}{(d+1) \cdot n},\end{aligned}$$
  where the second inequality follows that $M(n)$ has at most $(d+1) \cdot n$ tuples of proofs $(y_i, \pi_i)$, so the probability of choosing the right $k'$ given $\mathcal{A}'$ succeeds is
$$\frac{1}{l} \geq \frac{1}{(d+1) \cdot n}.$$

But such a $\mathcal{A}'$ violates theorem 28 for the underlying procedure, so its existence must not be the case, and so $\mathcal{A}$ cannot be finding the second-case solution (a solution not on the main computation line).

Thus, all there is left for such an $\mathcal{A}$ to exist is to solve the case one solution, which is finding a solution on the main computation line of
$$(S, V, (0^n, f(0^n), \emptyset), 2^n),$$
which would in turn solve the #SAT instance underlying $f$. That is, solving this constructed rSVL in ROM solves #SAT. ■

## 5.4 rSVL $\leq$ EoML and EoML $\in$ CLS

Now that #SAT $\leq$ rSVL in ROM assuming Fiat-Shamir heuristic being unambiguously sound, we show it to be in CLS. Since there are not very directly relevant to this literature, we only give a high-level idea.

**Definition 38** (End-of-Metered-Line, EoML)**.** *It is a tuple of three circuits $(S, P, M)$ where:*

- *$S$ is the successor circuit just like an* rSVL *instance.*
- *$P$ is the predecessor circuit that is supposed to reverse what $S$ does.*
- *$M : \{0,1\}^n \to [0, 2^n]$ is the meter circuit.*

*Given $0^n$, a source node, such that $M(0^n) = 1$ and $P(0^n) = 0^n \neq S(0^n)$, want to find one of the following:*

- *End of line: $P(S(v)) \neq v$ or $S(P(v)) \neq v \neq 0^n$ (this is the same as what EoL would search for without considering the new $M$ at all).*
- *False source: $v \neq 0^n$ and $M(v) = 1$. Source in the sense that meter says it's a starting point.*
- *Miscount: Either $M(v) > 0$ but $M(S(v)) - M(v) \neq 1$, or $M(v) > 1$ but $M(v) - M(P(v)) \neq 1$.*

**Lemma 39.** rSVL $\leq$ EoML.

*Proof.* It is the same idea as how SVL is reduced to EoML, which is by pebbling games. Pebbling game is a game where a pebble can only be put down on the $i$-th position if there is a pebble on the $(i-1)$-th position. Pebbling game is to show that $l$ pebbles are enough to put up to the $(2^l - 1)$-th position. The proof for this fact is a simple induction.

What does this do? Let's first inspect the reduction from SVL to EoML. Clearly, it would be sufficient already if we let the label of the EoML vertices keep track of the entire history of the previous steps on the SVL line, then it will obviously know how to compute the $P$ circuit as well as the $M$ circuit (just go to the previous step on the record for $P$ or count and check for $M$), while keeping the $S$ circuit the same. Issue is this is inefficient on an exponentially sized graph, so such a labeling needs to be and can be simplified using the idea of pebbling game. In more details: let $(S, V, T, 0^n)$ be any rSVL instance, we construct the following instance of EoML $= (S', P', M')$ (can be efficiently done starting from $0^n$ at every step):

- The starting node of EoML is $0^n$ by default, and it is labeled by $\emptyset$ in rSVL.
- Progressively from this $0^n$, every step we take, say we end up at $(v, i)$ at the succeeding step, we label $(v, i)$ by $\{(v_{i_j}, i_j)\}_{j=0}^{l}$ where $l \leq \log_2(T)$, because $v_j$ is a vertex on a position with pebble in them by the pebbling game at the current step (and $i_j$ is it's position). Note that these should only be valid if $V(v_{i_j}, i_j) = 1$ for all these $j$'s, otherwise $(v, i)$ should be a self-loop now, as it would be a solution for rSVL. This gives $S'$. We make all other instances not reachable this way self-loops.
- $P'$ circuit can be constructed by using the labels at a vertex to reverse the pebbling game to acquire the previous state.
- $M'$ can be constructed by using the label configuration to derive which step in the pebbling game the current vertex is at. (By this point is sufficient for the reduction from SVL, but we need to consider the additional false positive solutions for rSVL).

We analyze this reduction further to see why $(S', P', M')$ can work as a rSVL instance:

1. Every vertex is labeled by $\{(v_{i_j}, i_j)\}_{j=0}^{l}$ where $l \leq \log_2(T)$ (or they are self loops). **This is directly from construction.**
2. Every vertex that is not a self-loop, we know that all tuples in their label, $(v_{i_j}, i_j)$, are such that
$$V(v_{i_j}, i_j) = 1.$$
**This is directly from construction.**

3. All vertices on the main computation line must have labels that have

$$(v, i), S^i(v_0) = v.$$

As long as $u$ is not a self loop, it must have valid pebbling configurations (just because if it is not then we already made it a self loop). Thus, every predecessor and successor of every node on the main line of computation must adjust their pebbling configuration for their labels correctly.

Finally, we show the solutions found in EoML are solutions in the original rSVL: Let $u$ be a solution for the EoML instance. It either successfully reach the end of the original line of computation in rSVL correctly after $T$ steps with the correct pebbling configurations all the way through, or it is a self loop due to it's original instance in rSVL either was a self-loop to start with or violated the validity of $V$ (point 2 above means that there exists a vertex in the label of $u$ that is either the true sink ($V(v, T) = 1$) or a false positive; if it is a false positive then it must be off of the main line, which, by points 2 and 3 above, a false positive exists in its labels, which is poly-sized, so we can randomly choose one from the label and we will succeed with non-negligible probability). ■

**Proposition 40.** EoML $\in$ CLS. *This introduction, along with the definition of* EoML*, is shown in [HY20].*

## 5.5 Highlights

As section 7.6 pointed out, the line of work building up to PoSW raised a big open problem of uniqueness of proof procedures like IVC and PoSW. What [CHK+19] is able to show is, instead of constructing a proof process with such uniqueness, showing that computational uniqueness (unambiguity) is sufficient for constructing TFNP subclass average-case hardness. Such a proof procedure is called the incremental unambiguous computation. The very next section shows an alternative look on this, which is where the uniqueness of proofs like IVC and PoSW are attempted.

# 6 IVC construction #3 in ROM assuming iterated squaring is hard and inherent sequentiality (in turn factoring must be hard) with uniqueness [EFKP20]

Unlike [BG20] which uses the computational uniqueness in IVC to construct a hard instance of PLS instead of solving the stated lack of uniqueness problem of IVC / PoSW (or in this line of work in general; see section 7.6 for more about this open problem), [EFKP20] with its cVDF has a similar intention as iPoSW, but is actually targeted at solving the uniqueness problem (uniqueness is targeted at constructing a random beacon, but fits into the IVC with uniqueness needed for rSVL). Naturally, due to the similar requirements of a cVDF, it can also be used to construct hard instances of TFNP subclasses.

Note that by continuous, [EFKP20] means the same capability as the goal of an incrementally verifiable proof (i.e. a VDF whose computation can be computed from an arbitrary starting point in the middle of a full computation and ensure the same overall property). In other words, it can be more familiarly described as an incremental VDF in our context.

## 6.1 Verifiable, sequential and iteratively sequential functions

**Definition 41** ($B$-sound verifiable functions)**.** *Let $B : \mathbb{N} \to \mathbb{N}$. A $B$-sound verifiable function is a tuple of algorithms* (Gen, Eval, Verify) *where* Gen *is PPT,* Eval *is deterministic, and* Verify *is deterministic poly-time. The following properties are satisfied:*

- ***Perfect completeness.*** $\forall \lambda \in \mathbb{N}, pp \in \mathsf{Gen}(1^\lambda), x \in \{0,1\}^*$, *it holds that*

$$\mathsf{Verify}(1^\lambda, pp, x, \mathsf{Eval}(1^\lambda, pp, x)) = 1.$$

- *$B$-**soundness.** $\forall$ poly$(B(\lambda))$-sized circuit family $\{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$, $\exists$ a negligible* negl *such that $\forall\lambda \in \mathbb{N}$ it holds that*

$$\Pr_{pp\leftarrow\mathsf{Gen}(1^\lambda),(x,y)\leftarrow\mathcal{A}_\lambda(pp)}\left[\mathsf{Verify}(1^\lambda, pp, x, y) = 1 \text{ and } \mathsf{Eval}(1^\lambda, pp, x) \neq y\right] \leq \mathsf{negl}(\lambda).$$

*In other words, $B$-sound verifiable function is one where verification of the function's output can be done efficiently with completeness, but any family of circuits with size up to poly$(B(\lambda))$ has to either fine the correct $y$ consistent with* Eval *or verification will fail with very high probability if the $y$ that the adversary provides is inconsistent with* Eval*.*

**Definition 42** (Sequential function)**.** *Let $D, B, l : \mathbb{N} \to \mathbb{N}$ and let $\epsilon \in (0, 1)$. A $(D, B, l, \epsilon)$-sequential function is a tuple* (Gen, Sample, Eval) *where* Gen *and* Sample *are PPT and* Eval *is deterministic, and the following properties hold:*

- ***Honest evaluation.*** *Let $\{C_{\lambda,t}\}_{\lambda,t\in\mathbb{N}}$ be a uniform circuit family where $C_{\lambda,t}$ computes* $\mathsf{Eval}(1^\lambda, \cdot, (\cdot, t))$*. For sufficiently large $\lambda \in \mathbb{N}$ and $D(\lambda) \leq t \leq B(\lambda)$, we have*

$$\mathsf{depth}(C_{\lambda,t}) = t \cdot l(\lambda), \mathsf{width}(C_{\lambda,t}) \in poly(|\chi|).$$

- ***Sequentiality.*** *For any non-uniformly family of poly$(B(\lambda))$-sized circuits $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$, $\exists$negl, $\forall\lambda \in \mathbb{N}$, such that, if we sample $pp \leftarrow \mathsf{Gen}(1^\lambda), \mathcal{A}_1 \leftarrow \mathcal{A}_{0,\lambda}(pp), x \leftarrow \mathsf{Sample}(1^\lambda, pp), (t, y) \leftarrow \mathcal{A}_1(x)$, we have*

$$\Pr_{pp,\mathcal{A}_1,x,(t,y)}[\mathsf{Eval}(1^\lambda, pp, (x, t)) = y \text{ and } depth(\mathcal{A}_1) \leq (1 - \epsilon) \cdot t \cdot l(\lambda) \text{ and } t \geq D(\lambda)] \leq \mathsf{negl}(\lambda).$$

*In words, a sequential function is one in which the* Eval *function should be efficiently computable (by a poly-depth uniform circuit family) but an adversarial who tries to beat this honest evaluation depth has a negligible chance of getting the correct output (the $t$ multiplier must be greater than $D(\lambda)$).*

Then, the definition for cVDF follows from the two definitions above, as follows:

**Definition 43** (continuous verifiable delay function)**.** *Let $B, l : \mathbb{N} \to \mathbb{N}$ and $\epsilon \in (0, 1)$. A $(B, l, \epsilon)$-cVDF is a tuple* (Gen, Sample, Eval, Verify) *such that*

- (Gen, Sample, Eval) *is a $(1, B, l, \epsilon)$-iteratively sequential function.*
- (Gen, $\mathsf{Eval}^{(\cdot)}$, Verify) *is a $B$-sound function, and it satisfies the following completeness property.*
- *Furthermore, it satisfies **completeness from honest start**: for $\lambda \in \mathbb{N}$, $pp \leftarrow supp(\mathsf{Gen}(1^\lambda))$, $v_0 \in supp(\mathsf{Sample}(1^\lambda, pp))$, and $T \in \mathbb{N}$, it holds that*

$$\mathsf{Verify}(1^\lambda, pp, (v_0, T), \mathsf{Eval}^{(T)}(1^\lambda, pp, v_0)) = 1.$$

## 6.2 Repeated squaring assumption (aka the RSW assumption for Rivest-Shamir-Wagner, originally used to construct time-lock puzzles)

This is the essential hardness assumption, that iterated squaring is sequential in nature. More technically:

**Definition 44** (RSW assumption)**.** *Let $D, B : \mathbb{N} \to \mathbb{N}$. The $(D, B)$-RSW assumption is that $\exists$ polynomial $l \in \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$ such that RSW is a $(D, B, l, \epsilon)$-iteratively sequential function.*

**Lemma 45.** *RSW $\implies$ factoring is hard, as if factoring is not hard iterated squaring can be much expedited by factoring $N$ which the iterated squaring takes modulus over.*

## 6.3 A failed attempt

Recall the definition of a cVDF in definition 43, the following protocol is immediate:

- Say $\log B$ is the number of squarings we want to do.
- Compute the following in parallel: $x^T$ at $T = 1, 2, 4, \ldots, 2^{\log B}$, and have a proof for each one of them
- For any $i \in [T]$ where we want to get a value of $x_i = x_0^{2^i}$ at as well as the proof for it. Then, we can just pick the value from $T = 1, 2, \ldots, \log B$ that sum up to $i$, which is always possible (for example $11 = 8 + 2 + 1$, so joining $\pi_{0\to 8}, \pi_{8\to 19}$, and $\pi_{10\to 11}$ suffices).

Turns out, completeness fails. For instance, the proof for two steps for $\pi_{8\to 10}$ depends on a different internal state configuration than that for two steps for $\pi_{0\to 2}$, as an example. This is where the "uniqueness" in the sense of what IVC was missing is important. In particular, if we can make the internal state unique (equivalently, no internal state) for each step of the computation, each can be verified. That requires the prover strategy to be unique.

## 6.4 cVDF as modified from the Pietrzak's protocol construction

Like that of definition 25, we define a language on which an interactive proof system is defined. Here, the language is:

**Definition 46** ($\mathcal{L}_{N,B}$, Naive)**.** *Let $N = p \cdot q$ where $p$ and $q$ are safe primes ($p$ is a safe prime if $p = 2p' + 1$ where $p'$ is also a prime). Let $B$ be the upper bound of the number of times the squaring operation is taken:*

$$\mathcal{L}_{N,B} = \{(x, y, t) \mid x, y \in \mathbb{Z}_N^* \quad \textit{and} \quad y = x^{2^t} \pmod{N} \quad \textit{and} \quad t \le B\}.$$

*Note that $B$ is upper bounded by the time it takes to factorize $N$, otherwise it's trivial.*

Notice that $t$ squarings can be split into $k$ segments in the following sense. Suppose what we want to prove is $y = \left(x^2\right)^t$, so the computation can be split into the following segments:

$$x_1 \triangleq x^{2^{t/k}}, \ldots, x_i \triangleq x^{2^{it/k}}, \ldots, x_k \triangleq x^{2^{kt/k}} = \left(x^2\right)^t = y.$$

Note that squaring $x_i$ for $(t/k)$ times is exactly $x_i^{2^{t/k}}$, which is one incremental step of this protocol. That is,

$$x_i \text{ squared } t/k \text{ times} = x_i^{2^{t/k}} = \left(x^{2^{it/k}}\right)^{2^{t/k}} = x^{2^{it/k} \cdot 2^{t/k}} = x^{2^{(i+1)t/k}} = x_{i+1}.$$

**Why does this recursive process avoid the internal state lack of uniqueness problem?** The only internal state that the prover needs to maintain is the intermediate $u_i = x^{2^{it/k}}$ and the $y = x^{2^t}$. But, this requires proving that these $u_i$ and $y$ are computed correctly. Observe that the recursive process described above did indeed simplify the process of proving these $u_i$ and $y$. For example, if it takes $t$ squarings to prove $y$ in the first level, then it only requires $(t/k)$ squarings in the next level. This continues, until some $l$ levels later, the difficulty is only $t/k^l$ squarings.

### 6.4.1 Construction of cVDF

**Definition 47** (Sketch)**.** *Recall the definition that $x_i = x^{2^{it/k}}$ for some $i \in [k]$. Then, consider a random challenge that we sampled at every $t/k$ steps in the recursive layer, call it $r_i$ for $x_i$, then the following computations are equivalent:*

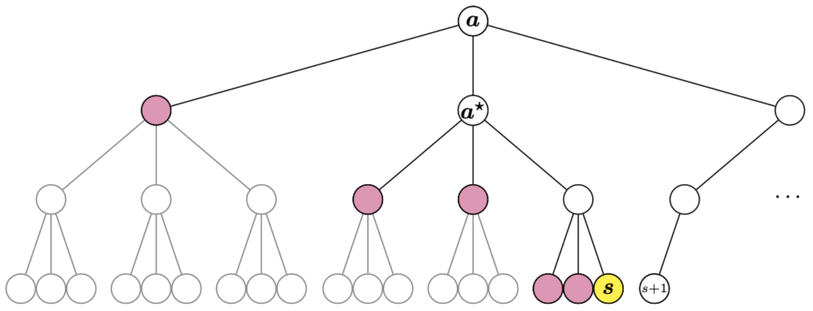
$$\left(\prod_{i=1}^{k} (x_{i-1})^{r_i}\right)^{2^{t/k}} = \prod_{i=1}^{k} \left((x^{2^{(i-1)t/k}})^{2^{t/k}}\right)^{r_i} = \prod_{i=1}^{k} \left(x^{2^{it/k}}\right)^{r_i} = \prod_{i=1}^{k} x_i^{r_i}.$$

*So, this final statement "$\prod_{i=1}^{k} x_i^{r_i}$" is true if and only if all of the $k$ segments each computing $t/k$ squarings are true. Such a combined statement is called a sketch.*

**Remark 48.** *The way to set $k$ is important for a refined argument, because if $k$ is proportional to $\lambda$, then $\log_k B \in O(1)$ for $B$ polynomial in $\lambda$, which gives a $O(1)$-round interactive proof protocol. This requires, technically, a weaker form of Fiat-Shamir heuristic to be made non-interactive than if $k$ itself is some constant independent from $\lambda$. Just note that this is one reason [EFKP20] seems to be a better construction than [CHK+ 19], but we largely overlook this difference.*

**Definition 49** (Frontier)**.** *This is essentially analogous to a path that needs to be recorded to hash for any leaf in the iPoSW construction that [BG20] used (see figure 2). Intuitively, the frontier is the set of nodes you need to compute the values at your current node. Concretely, the frontier of a node in the recursion tree ($k$-ary, directed, in our case) is the set of left siblings of nodes on the path from that node to root.*

**Example 50.** *For example, the set of the pink nodes is the frontier set of the yellow node, which is currently being computed.*



### 6.4.2 Strengthened version of $\mathcal{L}_{N,B}$ to ensure adaptive soundness, efficient verification, and completeness

Recall definition 46 for the memberships of the basic version of the language we want to prove with our protocol. It gave good intuitions, but now we add a few more restrictions so that the formal properties of a cVDF work.

**Definition 51** ($\mathcal{L}_{N,B}$, actual)**.** *Let $N = p \cdot q$ where $p, q$ are safe primes. Let $B = B(\lambda)$ be the upper bound of number of squarings we do. The,*

$$\mathcal{L}_{N,B} = \left\{ (x_0, y, t) \mid \begin{matrix} y^2 = (x_0^2)^{2^t} \pmod{N} \text{ if } x_0 \text{ is valid and } t \le B, \\ y = \bot, \text{ otherwise} \end{matrix} \right\},$$

*where we say that $x_0$ is a valid element if, after fixing any $N \in \mathbb{N}$,*

- $x \in \mathbb{Z}_N^*$,
- $\langle x^2 \rangle = \mathsf{QR}_N$ *(see definition 52), and*
- $x = |x|_N$.

*Similarly, we say that $(x_1, \dots, x_l)$ is a valid sequence if each $x_i$ is a valid element.*

### 6.4.3 Adaptive soundness intuition

This is a key part to the additional restrictions in definition 51, which is why we start treating $x_0^2$ and $y^2$ now, instead of the original numbers by themselves. It is due to some properties of $\mathsf{QR}_N$ that gives us adaptive soundness thorugh efficient proofs at each step.

**Definition 52** (Quadratic residue set $\mathsf{QR_N}$)**.** *$\mathsf{QR}_N \le \mathbb{Z}_N^*$ is a subgroup such that it contains all the quadratic residues of elements of $\mathbb{Z}_N^*$. That is, $\forall x \in \mathsf{QR}_N$, $x^2 \equiv \mu \in \pmod{N}$ for some $\mu \in \mathbb{Z}_N^*$.*

The following fact says that we have an efficient way to test of $\mu$ generates $\mathsf{QR}_N$ given the square root of $x^2 = \mu$. With this fact, the verifier can efficiently check that the start point of any computation is a valid generator of $\mathsf{QR}_N$.

For $N \in \mathbb{N}$, and any $x \in \mathbb{Z}_N$, we use the notation $|x|_N$ to denote $\min\{x, N - x\}$. Then,

**Fact 53.** *Let $N \in Supp(RSW.Gen(1^\lambda))$. Then, $\mu \in \mathbb{Z}_N^*$, it holds that $\langle\mu\rangle = \mathsf{QR}_N$ if and only if there exists an $x \in \mathbb{Z}_N^*$ such that $\mu = x^2$ and $\gcd(x \pm 1, N) = 1$.*

Using this fact, we know that $(x, y, t) \in \mathcal{L}_{N,B}$ if $(x^2)^{2^t} = y^2 \pmod N$. This is because $\mathsf{QR}_N$ is itself a group, so iterated squaring on an element of the group preserves its membership in the group. Thus, by checking an $x$ such that $x^2 \in \mathsf{QR}_N$ ensures soundness (no cheating adversary can choose a degenerate case). The point of doing this is that, once we start with an element in $\mathsf{QR_N}$, all the subsequent squared values will have this property, so it suffices for the verifier to check locally and efficiently ensure soundness of the protocol.

### 6.4.4 Interactive proof protocol for $\mathcal{L}_{N,B}$ and uVDF as a result

Throughout, even though $pp$ is the random seed, it is parsed into the following parameters:

$$pp = (N, B, k, d, hash).$$

We first describe a helper function, called $\mathsf{Sketch}$, that joins subgroups together to make incremental steps for a bigger proof higher up in the recursion tree:

**Algorithm 1** $\mathsf{Sketch}(pp, (x_0, t), y, msg)$

```
1: Parse $msg(x_1, \ldots, x_{k-1})$ and let $x_k = y$.
2: Let $(r_1, \ldots, r_k) = hash(pp, (x_0, t), y, msg)$   ▷ // relies on the fact ROM or some other assumption to give us a strong hash function so that we can make this setp non-interactive.
3: Let $x_0' = |\prod_{i=1}^k x_{i-1}^{r_i}|_N$ and $y' = |\prod_{i=1}^k x_i^{r_i}|_N$   ▷ // This reflects exactly one step in the sketch as described in definition 47.
4: if $x_0'$ is invalid then
5:     $y' \leftarrow \perp$
6: end if
7: return $(x_0', y')$
```

Using this, we define the prover and verifier for the interactive proof protocol:

- $\mathsf{Prover}(pp, (x_0, t), y)$:
    1. If $x_0$ is an invalid element, or if $t \leq k^d$ (base case), or $t > B$ (number of squarings exceed the bound), output $\perp$.
    2. Otherwise, $msg = (x_1, \ldots, x_{k-1})$ for
    $$x_i = \left|(x_0)^{2^{i \cdot t/k}}\right|_N.$$
    3. Compute $(x_0', y') = \mathsf{Sketch}(pp, (x_0, t), y, msg)$.
    4. Output $\pi = (msg, \pi')$ where $\pi' = \mathsf{Prover}(pp, (x_0', t/k), y')$
- $\mathsf{Verifier}(pp, (x_0, t), y, \pi)$:
    1. If $x_0$ is an invalid element, or $t > B$ (number of squarings exceed the bound), output 1 if $y = \pi = \perp$ and 0 otherwise (for invalid cases, only $y = \perp$ and, correspondingly, an empty proof can be accepted).

2. If $|y|_N$ is an invalid element, output 0 (this avoids trivializing the proof when $x$ is valid and all, but $|y|_N$ falls outside of the restrictions).
3. If $t \leq k^d$, output 1 if both $y^2 = (x_0)^{2^{t+1}} \pmod{N}$ and $\pi = \perp$ and output 0, otherwise (after $t$ has become small enough, the base-level proofs are computed directly).
4. Parse $\pi$ as $(msg, \pi')$, and output 0 if $msg$ is an invalid sequence.
5. Compute $(x_0', y') = \mathsf{Sketch}(pp, (x_0, t), y, msg)$.
6. Output $\mathsf{Verifier}(pp, (x_0', t/k), y', \pi')$.

The construction of the uniqueness VDF (uniqueness in the sense that, given $(x_0, t)$, it is sound in the sense that only one $y$ is likely to be verified) is the following:

- $[Gen(1^\lambda)]$. Sample $N = p \cdot q$ where $p, q$ are safe primes. Sample $hash \leftarrow \mathcal{H}$. Let $k = \lambda, B = B(\lambda)$, and $b$ be a constant that will be useful for technical reasons. Then, $p \triangleq (N, B, k, d, hash)$.
- $[Sample(1^\lambda, pp)]$. Sample $x_0 \leftarrow \mathbb{Z}_N^*$ at random so that $gcd(x_0 \pm 1, N) = 1$ and $x_0 = |x_0|_N$ (see section 6.4.3 why these specifications are useful).
- $[Eval(1^\lambda, pp, (x_0, t))]$.
  - $x_0$ is invalid, output $(y := \perp, \pi := \perp)$.
  - If $t \leq k^d$ (base case), output $(y, \perp)$, since $y$ is verified by direct computation, no proof is needed.
  - Otherwise, $x$ is valid, and a proof for the $y$ needs to be computed. So, we let
    $$x_i := |x_0^{i \cdot t/k}|_N, i \in [k],$$
    and let
    $$msg := (x_1, \ldots, x_{k-1}), y = x_k.$$
    Then,
    $$(x_0', y') = \mathsf{Sketch}(pp, (x_0, t), y, msg).$$
    Output $(y, \pi)$ where $\pi = (msg, \pi')$ and $\pi' = \mathsf{Prover}(pp, (x_0', t/k), y')$.
- $[Verify(1^\lambda, pp, (x_0, t), (y, \pi))]$.
  - Check for invalid $x_0$, or when $t$ exceeds the bound of $B$. In this case, proof is trivial, so let
    $$y = \pi = \perp \implies \text{output } 1; 0 \text{ otherwise}.$$
  - If $y$ is invalid, always return 0.
  - If both are valid, then return $\mathsf{Verifier}(pp, (x_0, t), y, \pi)$.

### 6.4.5 Claims about the constructed uVDF

We don't define uVDF here by itself, but instead point out its differences from cVDF in definition 43:

- We require a base difficulty level $D$, instead of 1. This is because uVDF is standalone, and it doesn't guarantee the capability of being verifiable at an intermediate step. In particular, difficulty $D$ for uVDF means that its an instance of a problem that requires at least $D$ squarings. However, difficulty 1 for cVDF just means that **a part of the problem** takes one squaring and this one squaring step by itself is also verifiable.
- We don't require **completeness from honest start**.

**Theorem 54.** *Let $D, B, \alpha : \mathbb{N} \to \mathbb{N}$ be functions satisfying $D(\lambda) \in \omega(\lambda^2), B(\lambda) \in 2^{O(\lambda)}$, and $\alpha(\lambda) \leq \lceil \log_\lambda(B(\lambda)) \rceil$. Suppose the following assumptions hold:*

- *$\alpha$-round strong FS assumption.*
- *$(D, B)$-RSW assumption for $l : \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$.*

*Then, $\forall \delta > 0$ and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$, it holds that the uVDF as constructed in section 6.4.4 is a $(D, B, (1 + \delta) \cdot l, \epsilon')$-uVDF.*

*Proof.* By definition, we have to show its completeness, soundness, and sequentiality. We state the results in as follows:

- **(Completeness).**

  **Lemma 55.** *For $\lambda \in \mathbb{N}, pp \in Supp(uVDF.Gen(1^\lambda)), x_0 \in \{0, 1\}^*, t \in \mathbb{N}$,*

  $$uVDF.Verify(1^\lambda, pp, (x_0, t), uVDF.Eval(1^\lambda, pp, (x_0, t))) = 1.$$

- **(Soundness).**

  **Lemma 56.** *The $(D, B, \epsilon)$-RSW assumption $\implies$ for every non-uniform algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ where $\mathcal{A}_\lambda$ runs in poly$(B(\lambda))$ time for every $\lambda \in \mathbb{N}$, $\exists$ negligible $\mathsf{negl}$ such that $\forall\lambda \in \mathbb{N}$:*

  $$\Pr\left[\begin{matrix} pp \leftarrow uVDF.Gen(1^\lambda) \\ (x_0, t), (\hat{y}, \hat{\pi}) \leftarrow \mathcal{A}_\lambda(pp) \end{matrix} : \begin{matrix} uVDF.Verify(1^\lambda, pp, (x_0, t), (\hat{y}, \hat{\pi})) = 1, \text{ and} \\ (\hat{y}, \hat{\pi}) \neq uVDF.Eval(1^\lambda, pp, (x_0, t)) \end{matrix}\right] \leq \mathsf{negl}(\lambda).$$

- **(Honest evaluation).**

  **Lemma 57.** *Suppose $(D, B)$-RSW for polynomial $l : \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$. Then, $\forall\delta > 0$, sufficiently large $\lambda \in \mathbb{N}$, and $t \geq D(\lambda)$, it holds that $uVDF.Eval(1^\lambda, \cdot, (\cdot, t))$ can be computed in time $(1 + \delta) \cdot t \cdot l(\lambda)$.*

- **(Sequentiality)**

  **Lemma 58.** *Suppose $(D, B)$-RSW for polynomial $l : \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$. Then, $\forall\delta > 0$, and $\epsilon' > \frac{\epsilon+\delta}{1+\delta}$, uVDF satisfies $(D, B, (1 + \delta) \cdot l, \epsilon')$-sequentiality.*

■

### 6.4.6 cVDF from uVDF

The previous construction of uVDF ensures that the final result that can be verified is unique, without guaranteeing anything about the intermediate steps. Next, we extend that construction to that of cVDF, such that intermediate steps can also be verifiable given honest start.

Recall definition 43 of cVDF. The key is that we want to make all the intermediate steps verifiable as well. What uVDF allows us to do is aggregation at every sketch node (which is the rightmost child of any parent node). The construction of cVDF is to leverage frontier set to allow intermediate proof and verification at any node. Here, we describe how the cVDF instance differs from the previous uVDF instance in each of the following aspects (Gen, Sample, Eval, Verify):

**Remark 59.** *The recursive structure is the same, and the difference is in the label. While a node for uVDF is $(x, y, t, \pi)$ which means $x^t = y$ with the proof $\pi$, a node for cVDF is*

$$(g, (s, \pi), F),$$

*where $g$ is the same as $x_0$ in uVDF, which is a starting element from $\mathbb{Z}_N^*$, and $s \in \{0, 1, \ldots, k\}^*$ is a path to a leaf node as a unique label in the most natural sense in a $(k + 1)$-ary tree (along with its proof $\pi$), and $F$ is the frontier set of node $s$.*

- **[$pp \leftarrow$ cVDF.$Gen(1^\lambda)$].** We use uVDF.$Gen(1^\lambda)$ to sample $pp_{\text{uVDF}}$. That gives us a $(pp_{\text{uVDF}}, d', g)$-puzzle tree, which is a $(k+1)$-ary tree where:
  - Each node is labeled $s \in \{0, 1, \dots, k\}^*$ (where the root node is the empty string).
  - Its child is $s \| i$ for $i \in \{0, 1, \dots, k\}$.
  - Its difficulty is $t = k^{h+d'+|s|}$. Note that $d'$ is a parameter, so that the height of the tree from root is $h = \lceil \log_k B \rceil - d'$.
  - Each node is also augmented by $\mathsf{val}(s) = (x, y, \pi)$, where $\pi = \text{uVDF}.\mathit{Eval}(pp_{\text{uVDF}}, (x, t))$.
- **[$v \leftarrow$ cVDF.$Sample(1^\lambda, pp)$].** Sample $g \triangleq x_0 \leftarrow \text{uVDF}.\mathit{Sample}(1^\lambda, pp_{\text{uVDF}})$ and output $v = (g, 0^h, \emptyset)$.

  **Remark 60.** *This $v$ points to the leftmost bottom level leaf node, which is why it is labeled by $0^h$. It requires no proof, as $g^{2^0} = g$ which is just itself.*

- **[$v' \leftarrow$ cVDF.$Eval(1^\lambda, pp)$].** Key changes happen here and in the $Verify$ function which comes next.
  - A node should be of the form $(g, s, F)$ ($\pi$ is absorbed in the notation as $s$). We want $s$ to be a leaf node in the $(pp_{\text{uVDF}}, g)$-puzzle tree, and $F$ is a frontier. Failure to parse causes $Eval$ to reject.
  - Run
  $$\text{cVDF}.Verify(1^\lambda, pp, \underbrace{((g, 0^h, \emptyset), s)}_{(v,T)}, \underbrace{(g, s, F)}_{v'}).$$
  If it rejects, $Eval$ should not continue and should reject too.
  - Now that the input is valid, we do the following:
    1. Compute the input of node $s$, which is done in one of the following cases:
       * **($s$ is a leftmost child):** recursively as the input passed down by parent.
       * **($s$ is a middle child):** input of node $s$ is the output of its left siblings' values (available in $F$).
       * **($s$ is a rightmost child):** Compute the sketch of its left siblings' values:
       $$\mathsf{Sketch}(pp_{uVDF}, (x, t), x_k, (x_1, \dots, x_{k-1})),$$
       all of which are available in $F$.
    2. Compute the value of $s$:
    $$(y, \pi) = \text{uVDF}.\mathit{Eval}(1^\lambda, pp_{\text{uVDF}}, (x, k^{d'})),$$
    where $x$ is the input acquired from the previous step.
    3. Compute the frontier of $s+1$ by merging partial proofs up to the closest common ancestor in the tree, so that the new frontier $(s+1, F')$ is correct and consistent (there are some complications depending on which of the three kinds of child $s$ is, but all are efficiently computable from the nearest common ancestor).

  Output $v' = (g, s+1, F')$ as the next node.

- **[$b \leftarrow$ cVDF.$Verify(1^\lambda, pp, (v, T), v')$].**
  1. Check if $v$ is well-formed as $(g, s, F)$ where $g \in \mathbb{Z}_N^*$, $s$ should be a leaf node, and $F$ is a frontier. If this parsing fails, reject. Otherwise, if $(g, s, F) \neq (g, 0^h, \emptyset)$, run
  $$\text{cVDF}.Verify(1^\lambda, pp, ((g, 0^h, \emptyset), s), (g, s, F)),$$
  and reject if this verification fails.

2. Check $v'$:
   (a) Parse $v'$ as $(g, s+T, F')$. $F'$ should be the frontier nodes in $\mathsf{frontier}(s')$ ($\mathsf{frontier}(s')$ can be efficiently found, because $|F'| \le h \cdot k \le (\lceil \log_k B \rceil - d') \cdot k)$, where $B$ is $poly(\lambda)$.
   (b) Check if $F'$ is consistent.
   **Definition 61** (Consistent)**.** *Let $S \triangleq \{(s, \mathsf{value})\}$, where $s$ is a node and* value *is its value in a $((N, B, k, d, hash), d', g)$-puzzle tree. Then, $(s', (x, y))$ is consistent with $S$ if*
      – *Input $x$ is correct based on one of the three kinds of child $s$ can be (see the first step of the third point of cVDF.Eval).*
      – *Output $y$ is correct by the third step of the third point of cVDF.Eval, which has two cases but we ignore the details.*

   *$S$ is said to be a consistent set if every node in $S$ is consistent with $S$.*
   (c) For $(s', (x, y, \pi)) \in F'$, run

$$\text{uVDF}.Verify(1^\lambda, pp_{\text{uVDF}}, (x, t\triangle k^{h+d'-|s'|}), (y, \pi)).$$

If any of these steps didn't succeed, reject. Otherwise, accept!

**Theorem 62.** *Let $D, B : \mathbb{N} \to \mathbb{N}$ where $B(\lambda) \le 2^{\lambda^{1/3}}$, $D(\lambda) = \lambda^{d'}$ for all $\lambda \in \mathbb{N}$ and specific constant $d'$. Assume*

- *$(D, B)$-RSW assumption for an $\epsilon \in (0, 1)$ and a polynomial $l$, and*
- *$\forall \epsilon', \delta \in (0, 1)$, uVDF is a $(D, B, (1+\delta) \cdot l, \epsilon')$-unique VDF.*

*Then, the resulting cVDF is a $(B, (1+\delta') \cdot D \cdot l, \epsilon'')$-cVDF for any $\epsilon'' > \frac{\epsilon+\delta'}{1+\delta'}$ and $\delta' > \delta$.*

## 6.5 Use cVDF to construct hard-on-average rSVL

[EFKP20] constructs a hard instance of rSVL (see definition 34) using cVDF, which implies hardness of TFNP subclasses as low as CLS.

**Theorem 63.** *Let $B, l, \epsilon : \mathbb{N} \to \mathbb{N}$ be functions where $B(\lambda) \ge \lambda^c$ for a sufficiently large $c$. If $\exists$ a $(B, l, \epsilon)$-cVDF, then* rSVL *is optimally HOA for algorithms with size $s(\lambda) \le (1-\epsilon) \cdot B(\lambda)$.*

*Proof.* Let (cVDF.*Gen*, cVDF.*Sample*, cVDF.*Eval*, cVDF.*Verify*) be a $(B, l, \epsilon)$-cVDF. The following is the construction for a HOA rSVL instance, $(S, V, T, v_0)$:

- **(Starting node $v_0$).** $pp \leftarrow$ cVDF.*Gen*$(1^\lambda)$ and then set $v_0 \leftarrow$ cVDF.*Sample*$(pp)$ .
- **(Difficulty $T$).** Let $T = \left(1 + \frac{\epsilon}{1-\epsilon}\right) \cdot s(\lambda)$.
- **(Successor circuit $S$).** Let
$$S(v) = \text{cVDF}.Eval(1^\lambda, pp, v).$$
- **(Verification circuit $V$).** Let
$$V(v, i) = \text{cVDF}.Verify(1^\lambda, pp, (v_0, i), i).$$

**(Totality of the instance).** Totality is guaranteed by the completeness of cVDF.

**(Hardness).** Suppose to the contrary that there is an efficient algorithm (with size, or parallel depth, at most $(1-\varepsilon) \cdot B(\lambda)$), $\mathcal{A}$, to solve this instance, then it must be able to solve it in one of two ways by definitioni:

1. $\mathcal{A}$ finds false positive $(v, i)$, so that

$$v \neq S^i(v_0) \quad \text{yet} \quad V(v, i) = 1.$$

By definition, this means that

$$v \neq \text{cVDF}.Eval^i(1^\lambda, pp, v_0) \quad \text{yet} \quad \text{cVDF}.Verify(1^\lambda, pp, (v_i, i), v) = 1.$$

This violates the soundness of cVDF.

2. $\mathcal{A}$ finds the actual sink $v$. That is to say that $\mathcal{A}$ finds

$$v = \text{cVDF}.Eval^{(T)}(v_0),$$

where $v_0$ is the initial state and $T \geq D(\lambda)$. Define the circuit $B_1$ as follows:

(a) **Hardcode cVDF parameters:** $B_1$ contains the public parameters pp used in the cVDF.

(b) **Simulate the adversary:** On input $v_0$, the circuit $B_1$ runs $\mathcal{A}$ internally to obtain $v$ such that cVDF.Verify$((v_0, T), v) = 1$.

(c) **Output:** The circuit $B_1$ finally outputs the pair $(v, T)$.

Since $B_1$ has size (or depth) bounded by that of $\mathcal{A}$, it follows that $B_1$ runs in strictly fewer than $T \cdot l(\lambda)$ steps (for some function $l$) on all inputs $v_0$. However, by iterative sequentiality of the cVDF, computing

$$(v, T) \leftarrow \text{cVDF}.Eval^{(T)}(v_0) \text{ s.t. } \text{cVDF}.Verify(1^\lambda, pp, (v_0, T), v) \text{ accepts}$$

cannot be done in fewer than $(1 - \varepsilon) \cdot T \cdot l(\lambda)$ sequential steps with high probability. This contradicts the iteratively sequential property of the cVDF.

So, $\mathcal{A}$ should be able to find neither of the solutions. ■

**Remark 64.** *As reasoned in the part of the survey for [CHK+19],* rSVL $\leq$ EoML $\in$ CLS. *So, under the assumption [EFKP20] makes, which are really:*

- *$\alpha$-round strong FS assumption, and*
- *$(D, B)$-RSW assumption for $l : \mathbb{N} \to \mathbb{N}$ and constant $\epsilon \in (0, 1)$,*

*there exists $(B, l, \epsilon)$-cVDF. This implies* rSVL *is HOA, which means everything above, including interesting classes like* CLS, PPAD, *etc, are HOA.*

## 7 IVC construction #4 in ROM assuming unconditionally without uniqueness [CP18, DLM19, BG20]

The CP graph as constructed in [CP18] can be thought of as an alternative to the graph in [MMV13]. In particular, instead of a general depth-robust graph only on the leaves level, it constructs a graph where new edges are created pointing to a leaf node from all nodes that are left siblings to any node on the path from that leaf node to the root node.

## 7.1 What did [MMV13] do?

The intuition is that [MMV13] uses a depth-robust DAG with $N$ vertices as a hardness assumption and lifts that to an efficiently verifiable proof protocol by building a Merkle tree on top of it, with each node of the graph as a leaf in the Merkle tree. For illustration, consider the following:

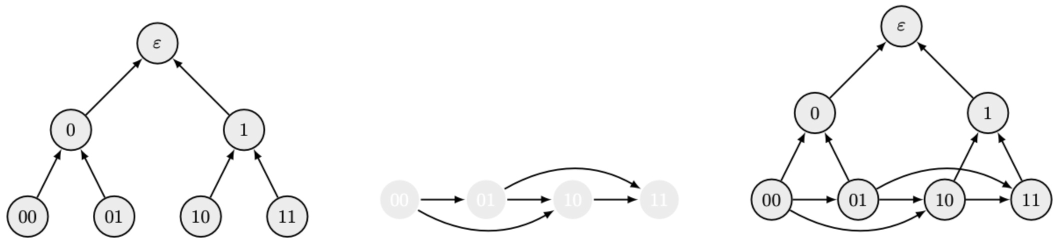


Figure 1: [MMV13] construction: (left) a Merkle tree, (middle) a toy depth-robust DAG with an unambiguous topological order, (right) each vertex of the DRG attached as a leaf of the Merkle tree from left to right in the topological order of the vertices.

## 7.2 The CP graph

[CP18] constructs a graph where new edges are created pointing to a leaf node from all nodes that are left siblings to any node on the path from that leaf node to the root node. See the following example:

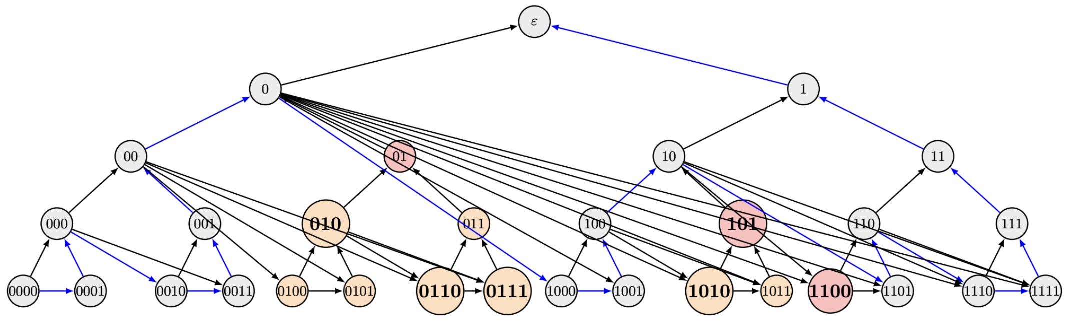


Figure 2: Example CP graph from [CP18]

Call such a graph $G_n^{PoSW}$ (where $n = 4$ in the illustrated example, as it has a height of 4). We state the following properties of such a graph proved in [CP18].

**Definition 65** (Labels of $G_n^{PoSW}$)**.** *Let $i$ be some vertex of $G_n^{PoSW}$, and $\{p_i\}_{i \in [d]}$ be its in-neighbors. Then, the label of $i$ is*

$$l_i = H(i, l_{p_1}, \dots, l_{p_d}).$$

*(Clearly $H$ in this context takes variables input-sizes, so we can consider it as a family of hash functions)*

Pictorially, by the order of hashing (and the topological order that needs to be preserved), the following is the labeling order:

**Lemma 66.** *The labels of $G_n^{PoSW}$ can be computed in topological order using only $w \cdot (n+1)$ bits of memory, where $w$ is the output length of the hash function $H$.*

*Proof.* Structural induction on $G_{n-1}^{PoSW}$ and the fact that:

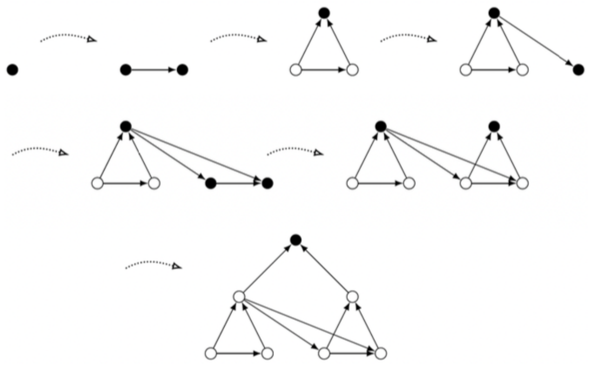


Figure 3: Borrowed from [BG20]

- **(Base).** $G_1^{PoSW}$: Hash the left and right nodes and hash them together once to the root node. In the most case, it holds in memory the hash of the left and right nodes (and a single bit root node name).
- **(Induction).** Left tree $G_{n-1}^{PoSW}$ is trivial, it is enough to just compute to $l_0$ as its label (since in the bigger $G_n^{PoSW}$, there are no new incoming edges to nodes in this subtree). And only remember $l_0$. Memory needed is $w \cdot n$.

  Right tree is a bit more complicated, as there are new in-edges from the node labeled as $l_0$ to all of its leaves. But the only extra thing needed is to hash this $l_0$ into the labels of the regular process of the right tree, meaning the additional memory needed is just $|l_0| = w$. The total memory needed is $w \cdot n + w = w \cdot (n+1)$. This should produce $l_1$ as the right tree's label.

  Finally, we just hash $H(\epsilon, l_0, l_1)$, with constant memory.

■

**Definition 67** (Special sets, $\hat{S}$, $S^*$, $D_S$)**.** *For $S \subseteq V$, we let*

- *$\hat{S}$ be all leave nodes below nodes in $S$.*
- *$S^*$ is the smallest set such that $\hat{S}^* = \hat{S}$.*
- *$D_S$ are all nodes in $S$ or below nodes in $S$.*

**Lemma 68.** *Let $S \subseteq V$. The subgraph of $G_n^{PoSW} = (V, E)$ induced on $V - D_{S^*}$ has a directed path going through all $|V| - |D_{S^*}| = N - |D_{S^*}|$ nodes.*

*Proof.* Induction on $n$. ■

**Lemma 69.** *For $S^*, S \subseteq V$, $D_{S^*}$ contains $|\{0,1\}^n \cap D_{S^*}| = \frac{|D_{S^*}|+|S^*|}{2}$ many leaves.*

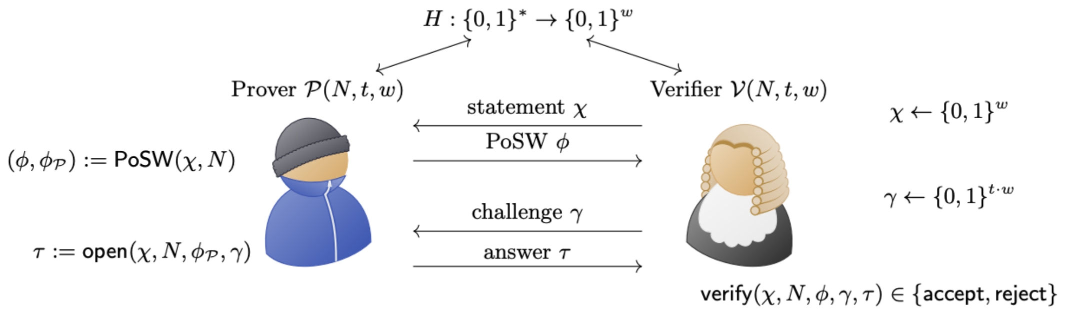
*Proof.* Trivial, just because $D_{v_i}$ is a full binary tree. ■

## 7.3 PoSW based on the CP graph

Let $n$ be the security parameter, $t$ be the statistical security parameter, and $N = 2^n - 1$. Let $\mathbf{H} : \{0,1\}^{w(n+1)} \to \{0,1\}^w$ be the hash function, instantiated as a perfect one in ROM. Let $M$ be memory available to $\mathcal{P}$, which is of the form $M = (t + n \cdot t + 1 + 2^{m-1}) \cdot w$ for some $0 \leq m \leq n$.

### 7.3.1 Construction

Proof of sequential work in both [MMV13] and [CP18] can be illustrated as follows:



$H : \{0,1\}^* \to \{0,1\}^w$
Prover $\mathcal{P}(N,t,w)$
Verifier $\mathcal{V}(N,t,w)$
statement $\chi$
PoSW $\phi$
challenge $\gamma$
answer $\tau$
$(\phi, \phi_{\mathcal{P}}) := \mathsf{PoSW}(\chi, N)$
$\tau := \mathsf{open}(\chi, N, \phi_{\mathcal{P}}, \gamma)$
$\chi \leftarrow \{0,1\}^w$
$\gamma \leftarrow \{0,1\}^{t \cdot w}$
$\mathsf{verify}(\chi, N, \phi, \gamma, \tau) \in \{\mathsf{accept}, \mathsf{reject}\}$

Let $\mathcal{X} \overset{\$}{\leftarrow} \mathbf{D}$ be the puzzle / statement to prove, and $(N, t, w)$ be the set of complexity / security parameters. As such, both $\mathcal{P}$ and $\mathcal{V}$ can be instantiated, and the hash function $\mathbf{H}_{\mathcal{X}}$ is fixed. The proof system then has the following three main components to specify:

- **(PoSW function).** On the $\mathcal{X}$ to prove, PoSW outputs the commitment $\phi$ (in [MMV13], this was the hashed labels on all $N$ vertices in order) and local computation record (in [MMV13], this was the local Merkle tree / look-up table).

  Here's what [CP18] does:

  1. Use $w(n+1)$ space to compute the labels of all nodes on $G_n^{PoSW}$ by lemma 66 using $\mathbf{H}_{\mathcal{X}}$.
  2. It sends the root label as the commitment:
  $$\phi = l_\epsilon.$$
  3. It stores the labels for top $m$ levels, i.e.
  $$\phi_{\mathcal{P}} = \{l_i\}_{i \in \{0,1\}^{\le m}}.$$

- **(Challenges).** $\gamma = (\gamma_1, \ldots, \gamma_t)$ are randomly sampled challenges (sampled only from leaves). It is of length $t \cdot w$ because each hash value has length $w$. So this is just **uniformly sampling**
$$\gamma \leftarrow \{0,1\}^{t \cdot w}.$$

- **(Open function).** On $\gamma$ challenge, for each $\gamma_i$ in it, for $i \in [t]$, open function contains in its output:
  - $l_{\gamma_i}$, label of the node in the challenge.
  - Labels of all siblings of the nodes on the path from $\gamma_i$ to the root.

  For example, if one of the $\gamma_i = 000$, based on the graph in figure 2, $\tau$ should o contain labels of $\{000, 001, 01, 1\}$.

  When $m = n$, no additional queries to $H_{\mathcal{X}}$ are needed.

- **(Verify).** By how $\tau$ is constructed, all labels of parents of the challenges, $\gamma_i$, are included in $\tau$. So, verify by checking if
$$l_{\gamma_i} = \mathbf{H}_{\mathcal{X}}(\gamma_i, l_{p_1}, \ldots, l_{p_d}), (l_{p_1}, \ldots, l_{p_d}) = \mathsf{parents}(\gamma_i).$$
  Now that we have $l_{\gamma_i}$, we then use labels in $\tau$ to recursively compute from $i = n-1$ to $i = 0$ and check if the final label value agrees with $l_\epsilon$, i.e. if
$$\mathbf{H}_{\mathcal{X}}(\gamma_i[0, \ldots, i], l_{\gamma_i[0,\ldots,i]\circ 0}, l_{\gamma_i[0,\ldots,i]\circ 1}) = l_\epsilon,$$
  which was committed previously as $\phi$ ($\gamma_i[0, \ldots, i]$ means recursively the first 0 bit of $\gamma_i$ up to the first $i$ bits).

### 7.3.2 Non-interactiveness in ROM

This is a pretty standard construction based on Fiat-Shamir in ROM, analogous to how it's explained in section ??. Again, this is because challenges are uniformly sampled, and the hash function is shared.

## 7.4 Proof of security

**Theorem 70.** *If a cheating prover $\tilde{P}$ makes fewer than $(1-\alpha)N$ sequential queries to the hash function $\mathbf{H}_{\mathcal{X}}$, the verifier accepts the proof with probability at most*

$$(1-\alpha)^t + \frac{2nwq^2}{2^w},$$

*which is negligible.*

*Proof.* Recall $\mathbf{H}$ is sequential and hash resistant. Let's call node $i$ inconsistent if $l'_i$, which is its actual computed label, does not agree with $l_i$ based on its parents. Let $S$ be the set of all inconsistent nodes.

By lemma 67, there exists a path going through all nodes $V - D_{S^*}$. Since $V - D_{S^*}$ are all consistent, the labels $\{l'_i\}_{i \text{ on this path}}$ constitute a $\mathbf{H}_{\mathcal{X}}$-sequence of length $(N - |D_{S^*}|)$. If $|D_{S^*}| \leq \alpha N$, then $\tilde{\mathcal{P}}$ must have made

$$\geq N - |D_{S^*}| \geq (1-\alpha)N \text{ sequential queries.}$$

So, in the case of making less than that many sequential queries, it must be the case that

$$|D_{S^*}| > \alpha N = \alpha(2^{n+1} - 1).$$

But, we also know from one of the lemmas above that

$$|\{0,1\}^n \cap D_{S^*}| = \frac{|D_{S^*}| + |S^*|}{2} > \alpha 2^n.$$

With this, we show that there is a high probability that one of the random challenges in $\gamma$ is going to intersect with one of the in consistent nodes to lead to a rejection, i.e.

$$\exists i \text{ s.t. } \exists v \text{ from the path from } \gamma_i \text{ to the root s.t. } v \in S.$$

As $\gamma_i$ is sampled uniformly randomly:

$$\Pr[\gamma_i \in D_{S^*}] = \frac{|\{0,1\}^n \cap D_{S^*}|}{2^n} > \alpha \implies \Pr[\gamma_i \notin D_{S^*}] < 1 - \alpha.$$

Union bound over $t$ random samples of $i$ gives

$$\Pr[\gamma \cap D_{S^*} = \emptyset] < (1-\alpha)^t,$$

so the verifier will reject $\tilde{\mathcal{P}}$ with probability greater than $1-(1-\alpha)^t$. ■

## 7.5 Efficiency

Here are some size parameters of inputs that are used in the three algorithms described in section 7.3.1: It is $w$ bits to specify any label and $n$ bits to specify any node:

$$|\mathcal{X}| = w, |\phi| = |l_\epsilon| = w, |\gamma| = t \cdot w, |\tau| \leq t \cdot w \cdot n.$$

After using Fiat-Shamir, we just send the commitment and the challenge response together, which needs a total memory of

$$|\phi| + |\tau| \leq w + t \cdot w \cdot n.$$

### 7.5.1 Prover efficiency

This is also a reason why it is valid for our use case of building hard $\mathsf{TFNP}$ subclass instances, because we need to be able to build a circuit (such as successor circuit) that simulates what prover does efficiently. As such, proof size, computation time, and memory must all be tractable.

- **($\mathbf{PoSW}^{\mathbf{H}_\mathcal{X}}(N)$).** For $N$ sequential queries total:

$$O(N \cdot (n+1) \cdot w)$$

- **($\mathbf{Open}^{\mathbf{H}_\mathcal{X}}(N, \phi, \gamma)$).** We can choose an $m \in [0, n]$ such that $\mathcal{P}$, when it solves the challenge, needs to recompute $2^{n-m+1} - 1$ many labels. There are $t$ challenges, for a total of

$$t \cdot \left(2^{n-m+1} - 1\right).$$

### 7.5.2 Verifier efficiency

The verifier does two things:

- sampling $t$ random challenges, and
- computing $\mathsf{Verify}(\mathcal{X}, N, \phi, \gamma, \tau)$, which is done by making $t \cdot n$ queries to $\mathbf{H}_\mathcal{X}$ (so just $O(t \cdot n)$ time), each with a length of $\leq n \cdot w$ (so for a total of $\leq t \cdot w \cdot n$ bits to communicate or write down).

## 7.6 Lack of uniqueness property

Note that this constructed protocol does not enforce uniqueness. That is, more than one proofs may be acceptable to the verifier under the given parameters (for example, an adversary may spend sufficient amount of time to come up with a proof that is not what an honest prover would come up with). By the time of [CP18], at least, it is not clear if such a uniqueness property can be guaranteed with the gap between proof generation and proof solving still being exponential.

Now, this uniqueness problem will cause problems in many of PoSW's applications. For us to show hardness in TFNP subclasses, this may mean that false positive solutions can be introduced that do not actually lead to a solution. [EFKP20] got around this problem by actually enforcing uniqueness, and [CHK$^+$19] did so by making false positive solutions due to lack of uniqueness computationally intractable to find (a property they call unambiguity). We will elaborate both of these late.

[DLM19] gives an even more space efficient construction of CP graph. The incremental concept is of our interest here so we briefly discuss its construction and note on some highlights. The construction of [DLM19] is incremental in the sense that a proof for time $N$ on $\mathcal{X}$ can be handed off and be topped for further computation, say another $N'$ steps, and teh final proof proves work for $(N + N')$ steps.

## 7.7 Construction tweaks

### 7.7.1 Why can't [CP18] construction be incremental?

Recall section 7.3.1, the challenges are sampled from the leaves of the $G_n^{PoSW}$ graph and determined by the root of this graph. Changing root changes the challenges so that the previous challenge set cannot keep working.

They key tweak of [DLM19] is choosing the challenge leaves on the flight at each node of the tree, and discard these challenges as the size of the tree grows. This way, further iterations of the tree only shave off root-to-leaf paths in the challenge set, as opposed to finding a completely new set of challenges. For illustration:

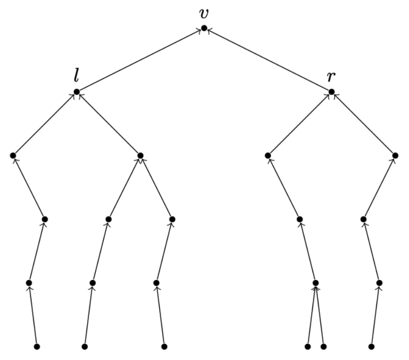


Fig. 1: Before choosing challenge subset.

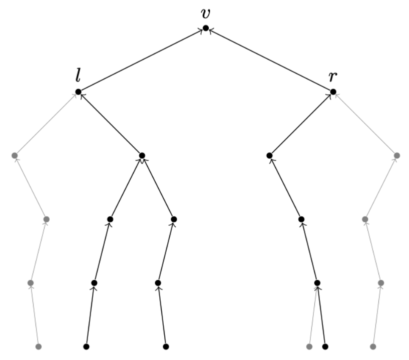


Fig. 2: After choosing challenge subset.

Note that due to the adaptivity of the adversary, just doing the above adaptive choice of challenge set paths leads to a $\log N$ loss in soundness (i.e. every time adaptivity happens, $\alpha$ soundness worsens to $\log N \cdot \alpha$; technical details see [DLM19]). The solution to this as proposed in [DLM19] is the following:

- by [CP18], left tree is computed before right tree is processed, so that the adaptivity can take advantage of the left tree to bias the right tree computation.
- So, instead, while doing the right tree computation, we also simultaneously recompute the left tree and fetch fresh challenge paths.

After all, when joining these two trees at a new parent node $v$, we choose $S_v$ which is a new set of $k$ challenges by choosing them at random from the $2k$ challenges in $S_l \cup S_r$ (made non-interactive by Fiat-Shamir).

### 7.7.2 Construction

Let $\chi \xleftarrow{\$} \mathcal{D}$ be the statement to prove, and $(N, t, \lambda)$ represent the complexity and security parameters, where $N$ is the time parameter, $t$ is the number of challenges, and $\lambda$ is the security parameter. Both the prover $\mathcal{P}$ and verifier $\mathcal{V}$ share access to the hash function $\mathbf{H}_{\mathcal{X}}$. The construction includes the following components:

- **(Prove function).** The prover computes the labels of nodes in $G_n^{\text{iPoSW}}$, similar to [CP18], but with additional features for incremental proof generation. Key steps are as follows:
  1. **Challenge Path Selection On-the-Fly:** At each node $v$, challenges are selected incrementally based on the random oracle $\mathcal{H}'_\chi$:
  $$r_v \leftarrow \mathcal{H}'_\chi(v), \quad S_v \leftarrow \text{RandomSubset}(2t, t; r_v),$$
  where $S_v$ determines the subset of challenge paths.
  2. **Efficient Memory Usage:** The prover maintains a list $U$ of *unfinished nodes*, storing paths for at most $\log N$ nodes at any point. This reduces the memory complexity to $O(\log N)$.
- **(Increment function).** To extend an existing proof $\pi$ for $\mathcal{X}$ with time $N$ to a proof for $N + N'$, the prover:
  1. Initializes from the root label and the stored challenge paths in $\pi$.
  2. Traverses the new portion of the graph $G_{n'}^{\text{iPoSW}}$, incrementally selecting challenges for the extended portion and discarding obsolete paths (that is, when merging, we randomly select $t$ challenges from the $t + t$ challenges from the two sub-trees).
- **(Verify function).** The verifier checks the proof as in [CP18] but adapted for incremental structure:

– For each challenge path, verify:

$$l_v = \mathcal{H}_\chi(v, l_{p_1}, \ldots, l_{p_d}), \quad \text{where } (p_1, \ldots, p_d) = \text{parents}(v).$$

– Ensure consistency between incremental segments and check that the updated proof aligns with the original commitment.

## 7.8 Parameter improvements

Compared to [CP18], [DLM19] achieves the following:

- **(Prover runtime).** The construction in [CP18] requires $N + \sqrt{N}$ sequential steps for the prover, leveraging a hybrid approach where $\sqrt{N}$ nodes are stored to reduce recomputation. The construction in [DLM19] achieves $N$ sequential steps for the prover by using incremental proofs and recomputation mechanisms.
- **(Memory complexity).** In [CP18], prover needs $\sqrt{N}$ memory, at the cost of additional computation. [DLM19] improved prover memory complexity to $\text{poly}(\log N)$, with a slight overhead in proof size (a factor of $\log^2(N)$ with single processor, and a factor of 9 with $\log(N)$ processors).
- **(Proof size).** In [CP18], proof size scales with the depth of the Merkle tree, which is $\mathcal{O}(\log N)$. [DLM19] worsened it by a factor of $\log^2(N)$ with single processor, and a factor of 9 with $\log(N)$ processors.
- **(Verifier runtime).** In [CP18], The verifier's runtime is $\textit{poly}(\log N)$, $\text{poly}(\log N)$. [DLM19] maintains this performance.

## 7.9 Construction of hard-on-average $\mathsf{PLS}^{\mathcal{O}}$ instances

So far, we have only talked about problems like $\mathsf{rSVL}$, $\mathsf{SVL}$, and $\mathsf{EoML}$, which are quite deep in the $\mathsf{TFNP}$ hierarchy, which means showing their hardness is at least as hard as showing classes that are higher up. As a recap, here are some key $\mathsf{TFNP}$ subclasses.

$$\begin{tikzcd}
 & & \mathsf{FNP} & & \\
 & & \mathsf{TFNP} \arrow[u] & & \\
\mathsf{PPA} \arrow[urr] & & \mathsf{PPP} \arrow[u] & & \mathsf{PLS} \arrow[ull] \\
 & \mathsf{PPAD} \arrow[ul] \arrow[ur] & & & \\
 & & \mathsf{CLS} \arrow[ul] \arrow[uurr] & & \\
 & & \mathsf{FP} \arrow[u] & &
\end{tikzcd}$$

So, it is a reasonable question to ask if there are simpler constructions / assumptions for hardness for higher-up classes than what [CHK$^+$19] has shown for $\mathsf{CLS}$-hardness (Fiat-Shamir + #SAT hardness in $\mathsf{ROM}$).

Indeed, [BG20] showed that $\mathsf{PLS}$ is *unconditionally hard* in $\mathsf{ROM}$, as it requires on iPoSW as described in [DLM19] based on [CP18], which can be instantiated in $\mathsf{ROM}$ (this works around [CHK$^+$19]'s use of IVC with computationally unique proofs to acquire SVL hardness). Here's a definition of polynomial local search, PLS:

**Definition 71.** *A PLS instance is defined by a two-tuple of poly-sized circuits, $(S, F)$, where:*

- *$S : \{0,1\}^n \to \{0,1\}^n$ is a successor circuit. Note that in-degree of a node is not guaranteed to be $\leq 1$.*
- *$F : \{0,1\}^n \to \mathbb{N}$.*

*A solution is a node $v \in \{0,1\}^n$ where $F(S(v)) < F(v)$. Equivalently, we construct a DAG, where $v$ is connected to $S(v)$ if and only if $F(S(v)) > F(v)$, and we want to find a sink ($F$ thus enforces a topological order that goes well with the iPoSW construction, and a solution is guaranteed by the totality of a finite DAG).*

**Definition 72** (Oracle-aided PLS)**.** *Let $S$ and $F$ both have access to an oracle.*

### 7.9.1 Construction overview

The iPoSW from [DLM19] in ROM is analogous to an IVC in the standard model, but different. They are different in the sense that the IVC construction of PLS hardness focuses on constructing some provably long computation, the hardness of the iPoSW construction comes from computing an accepting proof. Then, these hardness can be chained up across states by the fact that two accepting proofs in iPoSW can be efficiently merged to yield an overall accepting proof.

### 7.9.2 DLM iPoSW modified

The ideas as they were originally stated were in section 7.3.1 for a CP construction of PoSW, and in section 7.7.2 for a CP construction with incremental property. The key modification is that, on $G_n^{PoSW}$, instead of randomly sampling the challenges from the $2^n$ leaves, randomly sampling the new $k$ challenges from the $(k + k)$ challenges in the left and left subtrees, each being a $G_{n-1}^{PoSW}$. Rigorously, the prover algorithm is:

**Algorithm 2** DLM.P$_{\mathrm{H,\ H'}}(\mathcal{X}, t, \pi)$

**Input:**

- String $\mathcal{X} \in \{0,1\}^\lambda$
- Time $t \in [T]$
- Candidate proof $\pi = (\mathcal{L}, (u_i, S_{u_i}, P_{u_i})_{i=1}^m)$ // $\mathcal{L}$ is the current set of labels and $(u_i, S_{u_i}, P_{u_i})_{i=1}^m)$ where $u_i$ are currently active challenge vertices, where $S_{u_i}$ is its associated leaves and $P_{u_i}$ is the associated paths from these associated leaves to root. Note that these paths are sufficient to authenticate the labels of the challenge nodes up to this point.

**Algorithm:**

```
1: if $\{u_1, \dots, u_m\} \neq U_{t-1}$ then  // Lemma 6.2 of [BG20] states that $U_{t-1}$, which is the set of active vertices for each labeling of nodes in the topological order of the DLM construction, can be computed in $poly(\log N)$ where $N$ is the total number of indices for $U$.
2:     return $\epsilon$
3: end if
4: Set $\mathcal{L}[v_t] = H_\chi(v_t, \mathcal{L}[p_1], \dots, \mathcal{L}[p_r])$, where $(p_1, \dots, p_r) = \text{parents}(v_t)$ in topological order.
5: if $v_t$ is a leaf then
6:     Let $S_{v_t} = \{v_t\}$ and $P_{v_t} = \{(v_t)\}$
7: else
8:     // $v_t$ has two parents, $l, r \in U_{t-1}$, or really children in the typical binary tree terminology; in-neighbors as parents because of the directions of the edges.
9:     (a) Construct new challenge set. Sample a random subset $S_{v_t}$ of size $\min(s, |\text{leaves}(v_t)|)$ from $S_l \cup S_r$ using rand $\leftarrow H'_\chi(v_t, L[v_t])$ as public random coins.  // $s$ is a predetermined size of a challenge set. This is the key of making the typical CP graph construction an incremental computation.
10:    (b) Construct the new authentication path set. Let $P_{v_t} = \emptyset$. For every path in $P_l \cup P_r$, truncate them down to only the paths from leaves in $S_{v_t}$, and append these paths to $P_{v_t}$.  // i.e. all the nodes that are either the current challenge leaves or on the authentication path of the current challenge leaves.
11: end if
12: Remove labels from $L$ for all nodes but $\bigcup_{u \in U_t} P_u$.  // Again, $U_t$ can be computed efficiently in $poly(\log N)$.
13: return $\pi' = (L, (u, S_u, P_u)_{u \in U_t})$
```

As how it has been used, $H$ and $H$ in this prover protocol can be instantiated as perfect hash functions indexed by $\mathcal{X}$ with collision-resistant and sequential property in ROM (the two properties are proved in lemmas 4 and 5). They correspond to the public key randomized interactive proof protocol models, so that the verification interactions can be made non-interactive by Fiat-Shamir heuristic in ROM, see section 2.2.4.

**Remark 73** (Verifier)**.** *It should be intuitively clear why the authentication path and the challenge set is sufficient to convince a verifier that a certain number of steps have been taken (following collision resistance and sequentiality of the underlying hash functions, meaning that it is intractable for an adversary to cheat with a shorter time). As such, we don't write out the full verification algorithm in full details (see DLM.*$V^{H,H'}(\chi, t, \pi)$ *in [BG20]).*

**Remark 74** (Non-uniqueness)**.** *Challenges are incrementally randomly sampled by the prover. It is, thus, possible for the prover to choose a different set of challenges in each step to come up with a different proof of sequential work.*

### 7.9.3 Properties of the modified DLM construction

**Theorem 75** (Incremental completeness of modified DLM; proof for $t+1$ steps can be generated efficiently and correctly with proofs up to the $t$-th step)**.** *$\forall \lambda \in \mathbb{N}$, $t \in [T-1]$, candidate proof $\pi \in \{0,1\}^*$ and statement $\chi \in \{0,1\}^\lambda$: a new proof can be generated as $\pi' = DLM.P^{H,H'}(\chi, t, \pi)$, such that*

$$DLM.V^{H,H'}(\chi, t, \pi) = \mathsf{ACC} \implies DLM.V^{H,H'}(\chi, t+1, \pi') = \mathsf{ACC}.$$

*Proof.* The proof pretty much just analyzes different cases of the authentication paths and challenge nodes, and the argument follows from the iPoSW properties. In particular, the construction of the labeling algorithm of the iPoSW ensures that if the proof $\pi$ for a node $v$ is valid, the transition to the successor node $v'$ (either through advancing or merging sub-trees) generates a valid proof for $v'$. This property holds throughout, because the labeling algorithm is recursive and maintains consistency throughout the process of updating proofs. ■

**Theorem 76** (DLM soundness; no sub-exponential adversary can succeed)**.** *Let $\mathcal{A}^{H,H'} = \{\mathcal{A}_\lambda^{H,H'}\}$ be an oracle-aided adversary taht makes at most $q(\lambda)$ queries to both $H$ and $H'$. Then, $\forall \lambda, d \in \mathbb{N}$ with $d \leq \lambda$ such that $\mathcal{A}$ makes less than $2^d$ **sequential queries** to $H$, we have*

$$\Pr[DLM.V^{H,H'}(\chi, 2^{d+1}-1, \pi) = \mathsf{ACC} \mid \chi \leftarrow \{0,1\}^\lambda, \pi \leftarrow \mathcal{A}^{H,H'}(\chi)] \leq O(q^2)2^{-\Omega(\lambda)}.$$

*Proof.* Follows from sequential soundness property of PoSW construction in [CP18] and [DLM19]. ■

**Corollary 77** (Exponential hardness of finding an accepting proof)**.** *For every oracle-aided adversary $\mathcal{A}^{\mathcal{O}} = \{\mathcal{A}_\lambda^{\mathcal{O}}\}_{\lambda \in \mathbb{N}}$ of size $2^{o(\lambda)}$ and $\lambda \in \mathbb{N}$:*

$$\Pr[DLM.V^{\mathcal{O}}(\chi, 2^{\lambda+1}-1, \pi) = \mathsf{ACC} \mid \chi \leftarrow \{0,1\}^\lambda, \pi \leftarrow \mathcal{A}^{\mathcal{O}}(\chi)] \leq 2^{-\Omega(\lambda)}.$$

*Proof.* Follows form the latest theorem. ■

### 7.9.4 Hard $\mathsf{PLS}^{\mathcal{O}}$ instance, $R_d^{\mathcal{O}}$

Recall the verifier is $DLM.V^{H,H'}(\chi, t, \pi)$, which takes the statement to prove, time to be proved on and the proof for that amount of time to have elapsed sequentially, and efficiently decides between $\mathsf{ACC}$ / $\mathsf{REJ}$ with incremental completeness and soundness.

Now, let $t = T_d = 2^{d+1} - 1$. **We define the following search problem:**

**Definition 78** (Search problem $R_d^{\mathcal{O}}$)**.** *Let $d = d(\lambda) = \lambda$ be the depth parameter (where $\lambda$ is a computational security parameter). The search problem $R_d^{\mathcal{O}}$ is defined as*

$$R_d^{\mathcal{O}} = \{(\chi, \pi) \mid DLM.V^{\mathcal{O}}(\chi, T_d, \pi) = \mathsf{ACC}\}.$$

**We first show that $R_d^{\mathcal{O}}$ reduces to $\mathsf{PLS}^{\mathcal{O}}$.**

**Theorem 79.** $R_d^{\mathcal{O}} \in \mathsf{PLS}^{\mathcal{O}}$.

*Proof.* The idea is to have each node in the $\mathsf{PLS}^{\mathcal{O}}$ instance be $(t, \pi)$, and if it can be verified along with $\mathcal{X}$, i.e. $DLM.V(\chi, t, \pi) = \mathsf{ACC}$, that initiates the hash functions (hard-wired), then

- **($S^{\mathcal{O}}$).** the next state is $(t+1, \pi')$ where prover generates the new proof $\pi'$.
- **($V^{\mathcal{O}}$).** the value of the state is $t$.

Otherwise, if $(t, \pi)$ is $\mathsf{REJ}$ by the verifier, then

- **($S^{\mathcal{O}}$).** the next state is $(1, \pi_1)$.

- **($V^{\mathcal{O}}$).** the value of the state is $-1$.

By the efficiency and proof size of the process of DLM.P and DLM.V, $S$ and $V$ can be written down in $poly(|\chi|)$. In the end, we just output the proof part of the $(2^{d+1}-1, \pi)$ node as the solution, which is of length $poly(|\chi|)$.

Now, we show that if $(t, \pi)$ is a local maximum of $(S^{\mathcal{O}}, V^{\mathcal{O}})$, then

- $DLM.V^{\mathcal{O}}(\chi, t, \pi) = \mathsf{ACC}$, because if it's rejected, then, by the construction, the next state would be the beginning state which has value of 1, which is higher than the assigned value to the rejected state, $-1$, so it cannot be a solution.
- Now, for all $(t, \pi)$ that is accepted, by incremental completeness, there exists a proof $\pi'$ such that $DLM.V^{\mathcal{O}}(\chi, t+1, \pi') = \mathsf{ACC}$, too, if $t < T_d$.
- So, the only situation a local maximum can be found if we start from an accepting first state is $(T_d, \pi_f)$ where $\pi_f$ is the final associated proof.

As such, if $d(\lambda) \leq \lambda$, the above reduction finds a solution in an $\mathsf{PLS}^{\mathcal{O}}$ instance and the final $(T_d, \pi_f)$ gives the solution for $R_d^{\mathcal{O}}$, which is $\pi_f$. ■

**Finally, we show $R_d^{\mathcal{O}}$ is hard.**

**Theorem 80.** *$R_d^{\mathcal{O}}$ is exponentially hard on average. As such, $\mathsf{PLS}^{\mathcal{O}}$ has search problems that are hard-on-average.*

*Proof.* By setting $d = \lambda$, we have $R_\lambda^{\mathcal{O}} \in \mathsf{PLS}^{\mathcal{O}}$. By corollary 77, it is inverse exponentially hard to find a solution, so on average it is expected to take exponential time to solve this instance of $\mathsf{PLS}^{\mathcal{O}}$. ■

## 8 Directions

Possible directions: - Some easy things we have explicitly exercised. - Some directions we have tried that didn't work. - What's so important about uniqueness? To construct verifier circuit. Review complete problems in subclasses of TFNP that we care about. - What can we try further given the above summary (three constructions)? - cVDF seems to be in between the other two constructions. In particular, it doesn't use the direct facts about ROM, but instead goes through the iterated squaring assumption to show CLS hardness. Can this assumption be removed? - Unambiguity over iPoSW construction?

## References

[BG20] Nir Bitansky and Idan Gerichter. On the cryptographic hardness of local search. Cryptology ePrint Archive, 2020.

[CHK$^+$19] Arka Rai Choudhuri, Pavel Hubáček, Chethan Kamath, Krzysztof Pietrzak, Alon Rosen, and Guy N Rothblum. Finding a nash equilibrium is no easier than breaking fiat-shamir. In Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, pages 1103–1114, 2019.

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Annual international conference on the theory and applications of cryptographic techniques, pages 451–467. Springer, 2018.

[DLM19] Nico Döttling, Russell WF Lai, and Giulio Malavolta. Incremental proofs of sequential work. In Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II 38, pages 292–323. Springer, 2019.

[EFKP20] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 125–154. Springer, 2020.

[HY20] Pavel Hubácek and Eylon Yogev. Hardness of continuous local search: Query complexity and cryptographic lower bounds. SIAM Journal on Computing, 49(6):1128–1172, 2020.

[MMV11] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Time-lock puzzles in the random oracle model. In Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31, pages 39–50. Springer, 2011.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil Vadhan. Publicly verifiable proofs of sequential work. In Proceedings of the 4th conference on Innovations in Theoretical Computer Science, pages 373–388, 2013.

[RRR16] Omer Reingold, Guy N Rothblum, and Ron D Rothblum. Constant-round interactive proofs for delegating computation. In Proceedings of the forty-eighth annual ACM symposium on Theory of Computing, pages 49–62, 2016.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5, pages 1–18. Springer, 2008.