Technical Assessment – Carlo Brunetta

We expect you to send us within 7 days: a report answering the questions, the list of tools you have used, code you may have developed. Please send us your results in a zip file, along with your report in readable format (e.g. pdf) Don't hesitate to also include things that did not work, as experimentation is an important part of science! A Proof of Sequential Work (PoSW) enables showing that some computation was going on for T time steps since some statement was received. A lattice-based construction of PoSW is given in [LM24].

1. Present in a high-level way the construction given in [LM24]. Explain the intuition behind it.
2. Fully specify the resulting protocol and prototype it with your favourite programming language/tool (e.g., Python, Sage, C++, Rust, ...).
3. Analyse the efficiency of the implementation in terms of memory, communication and computation requirements.
4. How to turn the above protocol into a non-interactive proof system? Write down the resulting system by fully specifying the different steps.
5. Present one or two applications of the previous protocol and/or system.

Report Organisation.

- `tex/` Folder containing all the TEX files to generate the report.
- `code/` Folder containing the implemented code.
  - `code/main.py` Main script to run the PoSW prototype.
  - `code/posw.py` PoSW implementation.
  - `code/ring.py` Ring and related functionalities implementation.
  - `code/debug.py` Unpolished testing/debugging script.
- `report.pdf` This file: the report as a PDF with table of contents:

I will add in grey-coloured boxes (like this one) personal notes, failed attempts or similar.
Anything that I would share with others when brainstorming/solving problems!

List of Tools. The report is written in LaTeX using an (evolving) personal template for quick notes and Grammarly always checking for typos. The bibliography is a mix of cryptobib and eprint's BibTeX entries. The code was first designed on pen-and-paper and later developed in Python. Some tedious coding parts (e.g. command-line interface and comments/structuring) are modified versions of generated code from ChatGPT, i.e. first generate the skeleton-code and later fill up with my comments/algorithms.

The construction correctness, analysis and other examples are all developed on pen-and-paper.

I might have misunderstood the meaning of the request. Sorry if this is the case!

1 Lai–Malavolta's Construction

Present in a high-level way the construction given in [LM24]. Explain the intuition behind it.

Lai–Malavolta [LM24, LM23] provide the instantiation of a Proof of Sequential Work (PoSW) which allows a prover to prove to a verifier that $\mathbf{y}$ is obtained from the correct computation of a function f_T on input $\mathbf{x}$ and that the computation requires T sequential computation of f, in other words:

$$\mathsf{f}_T(\mathbf{x}) = \underbrace{\mathsf{f}(\mathsf{f}(\cdots \mathsf{f}(\mathbf{x})))}_{T \text{ times}} = \mathbf{y}$$

The main concept of such a primitive is that to obtain $\mathbf{y}$ there is no "faster way" other than computing f_T thus implying the necessity to *work* for an amount of time dependent on computing T times a function *sequentially*. Obviously, the verifier should be able to verify the correct computation in a faster than T sequential computation with the help of the prover.

The primitive is a combination of two fundamental concepts: *(i)* the sequential computation can be *described as a linear system* (despite not being linear) where the solution can only be obtained by executing f_T; *(ii)* the linear system is *self-similar* (like a fractal) thus one can release a central solution and "fold/aggregate" the remaining solutions into the solution of a smaller problem with half the size.

Sequentiality as Linear System. The function f is in itself the composition of a linear map $\mathbf{A}$ and a non-linear component $\mathbf{G}^{-1}$, i.e. $\mathsf{f}(\mathbf{x}) = -\mathbf{A} \circ \mathbf{G}^{-1}(\mathbf{x})$, with two notable properties:

- f is a collision-resistant hash function which motivates all the security arguments;
- $\mathbf{G}$ has a linear representation.

This second property allows considering the partial evaluations $\mathbf{u}_0 = -\mathbf{G}^{-1}(\mathbf{x})$ ($\mathbf{G}(\mathbf{u}_0) = \mathbf{x}$), $\mathbf{A}(\mathbf{u}_0) = \mathbf{y}_1$ which can be comfortably described into a linear system. For example, $\mathsf{f}_2(\mathbf{x}) = \mathsf{f}(\mathsf{f}(\mathbf{x})) = \mathsf{f}(\mathbf{y}_1) = \mathbf{y}_2$ can be described as the linear systems which can be combined into a single one:

$$\begin{pmatrix}\mathbf{G}\\ \mathbf{A}\end{pmatrix} \cdot (\mathbf{u}_0) = \begin{pmatrix}-\mathbf{x}\\ \mathbf{y}_1\end{pmatrix} \text{ and } \begin{pmatrix}\mathbf{G}\\ \mathbf{A}\end{pmatrix} \cdot (\mathbf{u}_1) = \begin{pmatrix}-\mathbf{y}_1\\ \mathbf{y}_2\end{pmatrix} \Longrightarrow \begin{pmatrix}\mathbf{G} & \\ \mathbf{A} & \mathbf{G}\\ & \mathbf{A}\end{pmatrix} \cdot \begin{pmatrix}\mathbf{u}_0\\ \mathbf{u}_1\end{pmatrix} = \begin{pmatrix}-\mathbf{x}\\ \mathbf{y}_1 - \mathbf{y}_1\\ \mathbf{y}_2\end{pmatrix} = \begin{pmatrix}-\mathbf{x}\\ 0\\ \mathbf{y}_2\end{pmatrix}$$

Clearly, this trick can be extended to any amount T of evaluations since only $\mathbf{G}, \mathbf{A}, \mathbf{x}, \mathbf{y}_T$ are required to define the whole sequential evaluation which is described as a linear system where the partial evaluations $(\mathbf{u}_i)_0^{T-1}$ are known only to who (honestly) computes $\mathsf{f}_T(\mathbf{x})$.

Folding the Linear System. The next step is allowing an honest prover with the partial evaluations $(\mathbf{u}_i)_0^{T-1}$ to provide proof to a verifier and this is done via a protocol that proves the knowledge of the solution of the linear system defining the evaluation $\mathsf{f}_T(\mathbf{x}) = \mathbf{y}_T$.

The trick used reassembles the act of (vertically) "folding" the matrix in half and noticing that the linear system can be split into two smaller ones with the same linear mapping $\mathbf{A}_t$, for example,

$$\left(\begin{array}{cc|c|cc} \mathbf{G} & & & & \\ \mathbf{A} & \mathbf{G} & & & \\ & \mathbf{A} & \mathbf{G} & & \\ \hline & & \mathbf{A} & \mathbf{G} & \\ & & & \mathbf{A} & \mathbf{G} \\ & & & & \mathbf{A} \end{array}\right) \begin{pmatrix}\mathbf{u}_0\\ \mathbf{u}_1\\ \mathbf{u}_2\\ \mathbf{u}_3\\ \mathbf{u}_4\end{pmatrix} = \left(\begin{array}{c|c|c} \mathbf{A}_t & \mathbf{G} & \\ \hline & \mathbf{A} & \mathbf{A}_t \end{array}\right) \begin{pmatrix}\mathbf{u}_0\\ \mathbf{u}_1\\ \mathbf{u}_2\\ \mathbf{u}_3\\ \mathbf{u}_4\end{pmatrix} = \begin{pmatrix}-\mathbf{x}\\ 0\\ 0\\ 0\\ \mathbf{y}_4\end{pmatrix}$$

If the prover provides the (central) value $\mathbf{u}_2$, the remaining solutions relates to the same linear system $\mathbf{A}_t$. By allowing the verifier to provide a random element r, the prover aggregates the solutions into one,

$$\mathbf{A}_t \cdot \begin{pmatrix}\mathbf{u}_0 + r \cdot \mathbf{u}_3\\ \mathbf{u}_1 + r \cdot \mathbf{u}_4\end{pmatrix} = \begin{pmatrix}-\mathbf{x}_0 - \mathbf{A}(\mathbf{u}_2)\\ \mathbf{y}_4 - \mathbf{G}(\mathbf{u}_2)\end{pmatrix}$$

In $\log(T)$ steps, the prover is left to share the final value $\mathbf{u}$ obtained by all the folded coefficients rs.

Personal Notes: the concepts underlying the construction are a mix of a paper of mine ([BLM19], code-based VRF) and Hypernova ([KS23], folding proof of knowledge). I love the folding trick! Sadly, I never thought/discovered/read about it until recently.

2 Protocol and Prototype

Fully specify the resulting protocol and prototype it with your favourite programming language/tool.

Quick Notation. Uniform sampling is denoted as $x \leftarrow\$ X$. Let $\mathcal{R}$ a ring of integer of a cyclotomic field and f denotes the degree of the extension (instead of ϕ used in the paper), vectors $\mathbf{a}$ have bold typesetting and matrices $\mathbf{A}$ are uppercase. For $q \in \mathbb{N}$, let $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ and $\mathcal{R}^\times$ denote the set of units of $\mathcal{R}$. A set $S \subset \mathcal{R}$ is subtractive if $\forall\, a, b \in \mathcal{R}$ with $a \neq b$, it holds $(a-b) \in \mathcal{R}^\times$. The inverse gadget matrix $\mathbf{G}^{-1}(\mathbf{x})$ returns the vector obtained by appending each p-ary representation for each entry of $\mathbf{x}$, i.e. for l such that $q < p^l$, $x = \sum_{j=0}^{l} x_j p^j$ and all $x_j < p$. Define the gadget matrix as $\mathbf{G} = I_n \otimes \begin{pmatrix} 1 & p & \cdots & p^l \end{pmatrix}$ which, briefly, takes a p-ary representation and returns the original value.

I highlighted in orange two objects that were somehow not present in the protocol's description. However, without them, the protocol would be ill-defined/weird or unusable, mainly because either the value might be hard to compute or prover–verifier might not agree on the same set.
I highlighted in red a typo present in the eprint version of the paper [LM24]. I'm unable to get the published version to check if the typo is present there too.

2.1 Lai–Malavolta **PoSW** Protocol

The whole PoSW's protocol is defined as:

1. initialise the public parameters $\mathsf{pp} \leftarrow \mathsf{Init}(\lambda)$
2. generate the challenge $(\mathbf{A}, \mathbf{x}) \leftarrow \mathsf{Gen}(\mathsf{pp})$
3. provide the challenge to the prover $\mathcal{P}$ which executes $\mathsf{Eval}(\mathbf{A}, \mathbf{x}, T)$ for some amount T, stores the witness ω and outputs $\mathbf{y}$
4. execute $\frac{\lambda}{\log \lambda}$ times the verification protocol with inputs $\mathsf{P}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, p, T), \omega)$ and $\mathsf{V}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, p, T)$
5. if the protocol never outputs bad, the verification protocol outputs valid. Otherwise, it outputs fail.

Let me expand on the different algorithms and how they are defined.

2.2 Construction Definition

The primitive is defined with the following algorithms $\mathsf{Init}, \mathsf{Gen}, \mathsf{Eval}$ plus an interactive protocol between a prover and a verifier. The algorithms are defined as follows:

- $\mathsf{Init}(\lambda) \to \mathsf{pp}$: initialise the public parameters pp containing the security parameter λ, the ring $\mathcal{R}$ of integers with a power-of-prime $\mathsf{poly}(\lambda)$-th root of unity, values q and p, dimensions n and m, subtractive set S, expansion value $\gamma_{\mathcal{R}}$.
- $\mathsf{Gen}(\mathsf{pp}) \to (\mathbf{A}, \mathbf{x})$: generate the challenge by uniformly sampling $\mathbf{A} \leftarrow\$ \mathcal{R}_q^{n \times m}$ and $\mathbf{x} \leftarrow\$ \mathcal{R}_q^n$
- $\mathsf{Eval}(\mathbf{A}, \mathbf{x}, T) \to (\mathbf{y}, \omega)$: for all $i \in [0, T-1]$ with $T > 0$ and $\mathbf{x}_0 = \mathbf{x}$, evaluate the function $\mathsf{f}(\mathbf{x}_i) = -\mathbf{A}\mathbf{G}^{-1}(\mathbf{x}_i) \pmod q = \mathbf{x}_{i+1}$ and store the values $\mathbf{u}_i = -\mathbf{G}^{-1}(\mathbf{x}_i)$. Output the witness $\omega = (\mathbf{u}_i)_{i=0}^{T-1}$ and evaluation result $\mathbf{y} = \mathbf{x}_T$.

The prover $\mathcal{P}$ and verifier $\mathcal{V}$ are involved into a recursive-like interactive protocol which can be defined using P, V for the prover and verifier's inputs respectively. Consider $\mathsf{P}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T), \omega)$ and $\mathsf{V}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T))$ where the public parameters pp are implicitly input to both the parties.

- if $T = 1$:
  - $\mathcal{P}$ sends $\mathbf{u}_0$ to $\mathcal{V}$
  - $\mathcal{V}$ receives $\mathbf{u}_0$
  - $\mathcal{V}$ outputs ok! if
$$\mathbf{u}_0 \in \mathcal{R}_\beta^m \bigwedge \mathbf{G}\mathbf{u}_0 = -\mathbf{x}; \bigwedge \mathbf{A}\mathbf{u}_0 = \mathbf{y}$$
    Otherwise, outputs bad.
- if $T = 2 \cdot t$:
  - $\mathcal{P}$ sends $\mathbf{u}_{T-1}$ to $\mathcal{V}$
  - $\mathcal{V}$ receives $\mathbf{u}_{T-1}$
  - verify that
$$\mathbf{u}_{T-1} \in \mathcal{R}_\beta^m; \bigwedge \mathbf{A}\mathbf{u}_{T-1} = \mathbf{y}$$
    If not, $\mathcal{V}$ outputs bad.
  - Repeat with
$$\mathsf{P}(\mathbf{A}, (\mathbf{x}, -\mathbf{G}\mathbf{u}_{T-1}, \beta, T-1), (\mathbf{u}_i)_{i=0}^{T-2}) \qquad \mathsf{V}(\mathbf{A}, (\mathbf{x}, -\mathbf{G}\mathbf{u}_{T-1}, \beta, T-1))$$
- if $T = 2 \cdot t + 1 > 1$:
  - $\mathcal{P}$ sends $\mathbf{u}_t$ to $\mathcal{V}$
  - $\mathcal{V}$ receives $\mathbf{u}_t$
  - verify that $\mathbf{u}_t \in \mathcal{R}_\beta^m$. If not, $\mathcal{V}$ outputs bad.
  - $\mathcal{V}$ samples $r \leftarrow\!\!\$\ S$ and sends it to $\mathcal{P}$
  - both computes:
    - $\mathbf{x}' = +\,\mathbf{x} + \mathbf{A} \cdot \mathbf{u}_t \cdot r \pmod q$
    - $\mathbf{y}' = \mathbf{y} \cdot r - \mathbf{G} \cdot \mathbf{u}_t \pmod q$
    - $\beta' = 2 \cdot \gamma_{\mathcal{R}} \cdot \beta$
  - $\mathcal{P}$ computes:
    - $\mathbf{u}_i' = \mathbf{u}_i + r \cdot \mathbf{u}_{t+1+i}$ for $i \in [0, t-1]$
  - Repeat with
$$\mathsf{P}(\mathbf{A}, (\mathbf{x}', \mathbf{y}', \beta', t), (\mathbf{u}_i')_{i=0}^{t-1}) \qquad \mathsf{V}(\mathbf{A}, (\mathbf{x}', \mathbf{y}', \beta', t))$$

Typo in the Paper. Previously highlighted in red a fixed typo from the paper. Consider the scenario $T = 2 \cdot t + 1$, the paper states $\mathbf{x}' = -\mathbf{x} - \mathbf{A} \cdot \mathbf{u}_t \cdot r \pmod q$. To see why this is a typo, consider $T = 3$.

After the first round of communication, the prover would compute $\mathbf{x}' = -\mathbf{x} - r\mathbf{x}_2$, $T' = 1$ and $\omega_0 = \mathbf{u}_0 + r\mathbf{u}_2$. At the next execution, the verifier $\mathcal{V}$ would compute,

$$\mathbf{G}\omega_0 = \mathbf{G}\mathbf{u}_0 + r\mathbf{u}_2 = \mathbf{G}(-\mathbf{G}^{-1}(\mathbf{x})) + r\mathbf{G}(-\mathbf{G}^{-1}(\mathbf{x}_2)) = -\mathbf{x} - r\mathbf{x}_2 \neq -\mathbf{x}' = \mathbf{x} + \mathbf{x}_2$$

which would produce a failing verification.

2.3 Prototype

I wrote the prototype in Python which code can be found in the `code/` folder.

Assumptions, Observations and Limitations.

- For the sake of simplicity, I wrote the prototype assuming the usage of the integers as the ring $\mathcal{R} = \mathbb{Z}$ meaning that $\gamma_{\mathcal{R}} = 1$ and $S = \{a, a+1\}$ for any $a \in \mathbb{Z}$. However, the code-class of the ring $\mathcal{R}$ is abstracted (enough) to allow a future plug-and-play usage of a more appropriate ring.
- When transforming into p-ary representation, $\mathbf{x}$ must be converted back into the range $[0, q)$.
- I fixed $m = n * l$ despite the paper suggesting the possibility of using different values.
- I developed the construction as defined (i.e. vertical vectors) despite this produces several matrix-transpositions calls in the code which affect memory and efficiency.

Run the Code! To execute the code, one can run:

```
# Default Values

python main.py
```

with (arbitrary) default values:

$$(p, q, \ a, n, \ T, \lambda \ , f, m, \ \text{tests}) = (19, 101, \ 0, 10 \ , 100, 1, \ 1, 1, \ 100)$$

Otherwise, the script allows modifying the parameters using the interface,

```
usage: main.py [-h] [--p P] [--q Q] [--a A] [--n N] [--T T] [--secpar SECPAR]
               [--f F] [--m M] [--tests TESTS]

PoSW Prototype using ZZ

optional arguments:
  -h, --help        show this help message and exit
  --p P             Base of the p-ary representation
  --q Q             Modulus of R_q
  --a A             Used to create the subtractive list
  --n N             Dimension of the PoSW
  --T T             Amount of sequential computations
  --secpar SECPAR   Security parameter (not used)
  --f F             f-th root of unity (order) (not used)
  --m M             Dimension of the PoSW (not used)
  --tests TESTS     Description for tests
```

The code outputs the passing rate, i.e. number of correct prove-verification executions, and some statistical values for both the evaluation Eval and prove-verify $\mathsf{P} + \mathsf{V}$ timing. For example, with the default values,

```
Passing Rate: 1.0

>>  Evaluation Time (ms)
Mean:       10.981431007385254
Variance:   0.0456968080243314
St. Dev:    0.21376811741775573
Median:     10.909080505371094

>>  Proving Time (ms)
Mean:       3.5293102264404297
Variance:   0.004362462959761615
St. Dev:    0.06604894366877956
Median:     3.5108327865600586
```

Observe: the proving time is smaller than the evaluation time, even for other parameters.
This is coherent with PoSW's goal!

3 Implementation Analysis

Analyse the implementation's efficiency: memory, communication and computation requirements.

Consider the operator $|x|$ that returns the size of the input x and for the rest of the section, $\log(x)$ denotes $\lceil \log_2(x) \rceil$, i.e. representation bits for x. For example, representing an element in $\mathbb{Z}_q$ costs $|\in \mathbb{Z}_q| = \log(q)$, the ring $|\mathcal{R}_q| = |q| + |f|$ with f being the degree of the ring extension and an element of the ring $|\in \mathcal{R}_q| = |q| \cdot f$.

Similarly, denote with the operator $\|x\|$ the computational cost of executing x, e.g. the value $\|\mathbf{A}\|$ is the cost of multiplying for a $\mathcal{R}_q^{n \times m}$ matrix, $\|+\mathbf{u}\|$ is the cost of summing two vectors of size $\mathbf{u}$ or $\|-\mathbf{x}\|$ for subtracting or changing the sing. For simplicity, the modulo $\pmod q$ is not explicitly stated but to be considered in the costs.

The section presents a theoretical analysis of the implementation space, computational and communication costs and some observations/suggestions for improvement.

Personal Note: I prefer studying efficiency on pen-and-paper, especially for primitives/protocols, mainly because measuring running code (sometimes) hides a lot of mysteries (e.g. interpreter/compiler optimizations, architecture-specific memory handling, literally magic).
I prefer measuring code timing whenever the analysis is focused on "real(istic) simulations" mainly coming from an application's requirement.

3.1 Space and Memory Costs

A mandatory object that both the prover and verifier have is PoSW's public parameters pp with size,

$$|\mathsf{pp}| = |\lambda| + |\mathcal{R}_q| + |p| + |n| + |m| + |\gamma_{\mathcal{R}}| + |S|$$

of which $|m| + |\gamma_{\mathcal{R}}|$ might be saved if they are quickly computable from $(n, \mathcal{R}_q, p)$. In the implementation and as for the prototype's assumptions, $(\lambda, m, \gamma_{\mathcal{R}})$ and f are fixed while S must be of the form $S = \{a, a+1\}$ thus the effective implementation cost is:

$$|\mathsf{pp}| = |\mathcal{R}_q| + |p| + |n| + |S| = |q| + |p| + |n| + |q|$$

Regarding the challenge, the cost is all in the representation of $(\mathbf{A}, \mathbf{x})$:

$$|\mathbf{A}| + |\mathbf{x}| = (n \cdot m + n) \cdot |\in \mathcal{R}_q|$$

The evaluation Eval algorithm outputs $\mathbf{y}$ and the witness ω with weights[1],

$$\begin{cases} |\mathbf{y}| + |\omega| = n \cdot |\in \mathcal{R}_q| + T \cdot |\mathbf{u}_i| \\ |\mathbf{u}_i| = m \cdot |p| \cdot f = n \cdot l \cdot |p| \cdot f \simeq n \cdot |q| \cdot f = n \cdot |\in \mathcal{R}_q| \end{cases}$$

for a total cost of

$$|\mathbf{y}| + |\omega| = n \cdot (T+1) \cdot |\in \mathcal{R}_q|$$

Regarding the computational memory cost while computing Eval, the prover must maintain a single register of size $|\mathbf{u}_i|$ where the results of the $\mathbf{G}^{-1}, \mathbf{A}$ evaluations are effectively computed.

For both the prover and verifier, the verification protocol must maintain a state for which memory costs have a maximum, for the prover, at the beginning and later reduce because of the folding/release of witnesses. At the start, the cost is,

$$|\mathbf{A}| + |\mathbf{x}| + |\mathbf{y}| + |T| + \underbrace{|\omega|}_{\text{prover}} = (n \cdot m + 2n) \cdot |\in \mathcal{R}_q| + |T| + \underbrace{n \cdot |\in \mathcal{R}_q|}_{\text{prover}}$$

[1] I used $\simeq$ because it might be that $q < p^l$ thus there is technically a small difference.

3.2 Computational Cost

The computational cost of Init is mainly storing/sampling the parameters dependent from λ while generating the challenge requires the sampling of $n \cdot m + n$ elements in $\mathcal{R}_q$. This algorithm's computational costs highly depend on the sampling $\|\leftarrow\$\, \mathcal{R}_q\|$ efficiency.

The effective computational cost depends on some basic operations, namely:

- $\mathbf{G}^{-1}$ action: transforming an element $\mathbf{u} \in \mathcal{R}_q$ in p-ary notation
- $\mathbf{G}$ action: transforming p-ary notation in an element $\mathbf{u} \in \mathcal{R}_q$, done by a matrix multiplication
- $\mathbf{A}$ action: matrix multiplication
- element-wise operations, e.g. checking equality of two vectors, sum or negate a vector
- sampling from S

The evaluation procedure executes T times a p-ary transformation, the negation of $\mathbf{u}$ and the multiplication $\mathbf{Au}$ with total cost,

$$T \cdot \left(\left\|\mathbf{G}^{-1}\right\| + \|-\mathbf{u}\| + \|\mathbf{A}\|\right)$$

Regarding proving and verifying, the computational cost depends on T which implies[2] a specific amount t_1 of no-foldings ($T = 2t$) and t_2 of foldings ($T = 2t+1$). The sum is lower bounded $t_1 + t_2 \leq \log T$ with equality when $t_1 = 0$, i.e. there are only foldings which corresponds to $T = 2^k - 1$ for some $k \in \mathbb{N}_{>0}$.

For the P algorithm, the prover must compute

$$\begin{aligned} &t_1 \cdot (\|\mathbf{G}\| + \|-\mathbf{x}\| + \|T-1\|) + \\ &+ t_2 \cdot (\|\mathbf{A}\| + \|r \cdot \mathbf{y}\| + \|+\mathbf{x}\| + \|r \cdot \mathbf{y}\| + \|-\mathbf{x}\| + \|\mathbf{G}\| + 2\,\|\cdot\beta\| + \|+\mathbf{u}\| + \|r \cdot \mathbf{u}\|) \end{aligned}$$

while the verifier has the checks as additional computations,

$$\begin{aligned} &1 \cdot \left(\left\|\in \mathcal{R}_\beta^m\right\| + \|\mathbf{G}\| + \|=\mathbf{x}\| + \|\mathbf{A}\| + \|=\mathbf{y}\|\right) + \\ &+ t_1 \cdot \left(\left\|\in \mathcal{R}_\beta^m\right\| + \|\mathbf{A}\| + \|=\mathbf{y}\| + \|\mathbf{G}\| + \|-\mathbf{x}\| + \|T-1\|\right) + \\ &+ t_2 \cdot \left(\|\leftarrow\$\, S\| + \left\|\in \mathcal{R}_\beta^m\right\| + \|\mathbf{A}\| + \|r \cdot \mathbf{y}\| + \|+\mathbf{x}\| + \|r \cdot \mathbf{y}\| + \|-\mathbf{x}\| + \|\mathbf{G}\| + 2\,\|\cdot\beta\|\right) \end{aligned}$$

The computation with highest cost is the matrix multiplication $\|\mathbf{A} \cdot \mathbf{u}\|$ which costs $m \cdot n = n^2 \cdot l$ multiplications (naively) in $\mathcal{R}_q$ for a total "rule-of-thumb" (i.e. asymptotic) cost of $\mathcal{O}\left(n^2 \cdot \log_p(q) \cdot \|\times\mathcal{R}_q\|\right)$ where all the parameters (seem to be) depend on $\mathsf{poly}(\lambda)$. The multiplication $\mathbf{G} \cdot \mathbf{u}$ takes only $m = n \cdot l$ field multiplication because of the sparse representation of $\mathbf{G}$, i.e. functionally it is a diagonal matrix. In asymptotic form, the total cost for the prover $\mathcal{P}$ and verifier $\mathcal{V}$ is:

$$\|\mathcal{P}\| = \|\mathsf{Eval}\| + \|\mathsf{P}\| \sim \mathcal{O}\left((T + t_2) \cdot n^2 \cdot \log_p(q) \cdot \|\times\mathcal{R}_q\|\right) \qquad \|\mathcal{V}\| \sim \mathcal{O}\left((1 + t_1 + t_2) \cdot n^2 \cdot \log_p(q) \cdot \|\times\mathcal{R}_q\|\right)$$

Minor Optimisation. As a possible computational optimization with some space cost, both the prover and verifier can pre-compute $(p^i)_{i=2}^{l}$ to halve the number of multiplications when computing $\mathbf{G}(\mathbf{u})$. The additional space cost would be $l \cdot |q|$.

I'm struggling to get a better/concrete complexity because the majority of parameters are in asymptotic format which would require a thorough analysis of SIS's (concrete) security which I couldn't quickly do and/or don't precisely know/remember at the moment.

[2] The total amount $t_1 + t_2$ of communication rounds is equal to the number of multiplications executed in the Chandah-sutra exponentiation method. Integer sequence: `https://oeis.org/A014701`

3.3 Communication

The initialization and challenge generation must share the public parameters pp, challenge $(\mathbf{A}.\mathbf{x})$ beforehand, with cost (from Section 3.1):

$$|\mathsf{pp}| + |\mathbf{A}| + |\mathbf{x}| = |\lambda| + |\mathcal{R}_q| + |p| + |n| + |m| + |\gamma_{\mathcal{R}}| + |S| + (n \cdot m + n) \cdot |\in \mathcal{R}_q|$$

I assume that the communication starts with either prover or verifier sharing the value T with communication cost $\log(T)$. From the protocol, it is easy to see that the prover sends $(1 + t_1 + t_2)$ values $\mathbf{u}_i$ while the verifier only replies with t_2 random value r. This implies a total communication cost of,

$$|\mathsf{P} + \mathsf{V}| = \log(T) + (1 + t_1 + t_2) \cdot |\mathbf{u}_i| + t_2 \cdot |r \in S| = \log(T) + (1 + t_1 + t_2) \cdot n \cdot |\in \mathcal{R}_q| + t_2 \cdot \log(\#S)$$

where $\#S$ indicates the cardinality of S since, despite the elements of S having size $|\in S| = \log(q)$, the primitive can uniquely index S thus only requiring to share the sampled index with size $|\#S| = \log(\#S)$.

4 Non-Interactive Transformation

Turn the protocol into a non-interactive proof system, and write the full specification.

To obtain a non-interactive proof system, I would suggest using the Fiat-Shamir (FS) transformation. In a nutshell, the prover substitutes requesting the verifier to sample and send a random challenge c with the output of a random oracle (RO) query on the current communication transcript. The key idea is that it is hard for the prover to predict the oracle's output thus forcing the prover to follow the protocol honestly since the verifier can re-compute the same values from the oracle too during verification. Additionally, the verifier is never queried thus obtaining the non-interactivity.

I'm not sure if I will have the time later but I tried to (very) quickly check some zero-knowledge papers (mainly [AFK21]) to see if there is a "safer" way to deal with this sort of "dependent multi-round + parallel challenges" but couldn't find anything pointing to a better methodology.
Therefore, I will go for the solution that *feels* more secure!

Sketch Idea. First, one must introduce a concrete instantiation for the RO. For the sake of simplicity, consider a cryptographic hash function H with codomain S which must be introduced during the instantiation of the public parameters pp.

The interactive protocol is split into the procedures for proving and verifying the evaluations which follows the protocol structure by reading/writing into a transcript τ instead of receiving/sending messages to the other party. As the name suggests, the transcript emulates the communication between the prover and verifier thus the prover can instantiate it with some initial description, e.g. the public parameters, challenge and result $\tau_0 = \mathsf{pp} \| \mathbf{A} \| \mathbf{x} \| \mathbf{y}$.

The prover starts executing $\mathsf{P}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, p, T), \omega)$ and each time it should send/receive a message, this is appended to τ_{i-1} to get τ_i. In particular, the received messages are all randomly sampled elements r_i which are computed by hashing the current transcript $r_i = \mathsf{H}(\tau_{i-1})$. At the end of the emulated communication, the prover outputs the final transcript as the non-interactive proof $\pi = \tau_{t_1+t_2}$.

The verifier received the proof π, starts from the same initial transcript τ_0 and sequentially reads the messages in the transcript. Since T is known, the verifier knows which are the random elements thus it can verify the correct execution by recomputing and verifying the correct usage of all the $r_i = \mathsf{H}(\tau_{i-1})$.

Using a cryptographic hash function H as an instantiation of a random oracle is sometimes arguable since it might not be the best choice for some applications.
Maybe a PRF and pushing the problem of getting a common random key/seed?
I'm choosing to not go (for now) into the rabbit hole and find out the most correct type of function!

4.1 Non-Interactive PoSW Primitive

Let me report the protocol of Section 2 and highlight in blue the addition made to transform the primitive into a non-interactive one. For the sake of readability, the algorithms $\mathsf{Eval}, \mathsf{Gen}$ are not reported since they are unaltered in the non-interactive version. The algorithms are defined as follows:

- $\mathsf{Init}(\lambda) \to \mathsf{pp}$: initialise the public parameters pp containing the security parameter λ, the ring $\mathcal{R}$ of integers with a power-of-prime $\mathsf{poly}(\lambda)$-th root of unity, values q and p, dimensions n and m, subtractive set S, expansion value $\gamma_{\mathcal{R}}$ and secure hash function H (representing the random oracle).
- $\mathsf{P}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T), \omega) \to \pi$: the proving procedure is computed recursively by calling the procedure $\overline{\mathsf{P}}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T), \omega, \tau)$ with starting transcript $\tau = \mathsf{pp} \| \mathbf{A} \| \mathbf{x} \| \mathbf{y} \| R$ where R is a uniformly random bit-string. The procedure is defined as follows:
  - if $T = 1$:
    - the procedure appends $\mathbf{u}_0$ to τ
    - output the final transcript τ
  - if $T = 2 \cdot t$:
    - the procedure appends $\mathbf{u}_{T-1}$ to τ and obtains τ'
    - recursively execute $\overline{\mathsf{P}}(\mathbf{A}, (\mathbf{x}, -\mathbf{G}\mathbf{u}_{T-1}, \beta, T-1), (\mathbf{u}_i)_{i=0}^{T-2}, \tau')$
  - if $T = 2 \cdot t + 1 > 1$:
    - the procedure appends $\mathbf{u}_t$ to τ and obtains τ'
    - the procedure computes $r = \mathsf{H}(\tau')$ and appends it to τ' and obtains τ''
    - the procedure computes:
      - $\mathbf{x}' = \mathbf{x} + \mathbf{A} \cdot \mathbf{u}_t \cdot r \pmod q$
      - $\mathbf{y}' = \mathbf{y} \cdot r - \mathbf{G} \cdot \mathbf{u}_t \pmod q$
      - $\beta' = 2 \cdot \gamma_{\mathcal{R}} \cdot \beta$
      - $\mathbf{u}'_i = \mathbf{u}_i + r \cdot \mathbf{u}_{t+1+i}$ for $i \in [0, t-1]$
    - recursively execute $\overline{\mathsf{P}}(\mathbf{A}, (\mathbf{x}', \mathbf{y}', \beta', t), (\mathbf{u}'_i)_{i=0}^{t-1}, \tau'')$

  At the end of the recursive execution, the final transcript is the outputted proof $\pi \leftarrow \tau$.
- $\mathsf{V}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T), \pi) \to \{\mathsf{valid}, \mathsf{fail}\}$: the verifying procedure is computed recursively by calling the procedure $\overline{\mathsf{V}}(\mathbf{A}, (\mathbf{x}, \mathbf{y}, \beta, T), \omega, \pi, \tau)$ with starting transcript $\tau = \mathsf{pp} \| \mathbf{A} \| \mathbf{x} \| \mathbf{y} \| R$ where R is a uniformly random bit-string. For readability, reading from the proof *consumes* the proof π (similarly to an iterator) and appends the content to the transcript τ (and adds a $\prime$ in the notation, e.g. from $\tau \to \tau' \to \tau''$). The procedure is defined as follows:
  - if $T = 1$:
    - the procedure reads $\mathbf{u}_0$ from π
    - the procedure outputs $\mathsf{ok!}$ if $\mathbf{u}_0 \in \mathcal{R}_\beta^m$, $\mathbf{G}\mathbf{u}_0 = -\mathbf{x}$ and $\mathbf{A}\mathbf{u}_0 = \mathbf{y}$. Otherwise, outputs bad.
  - if $T = 2 \cdot t$:
    - the procedure reads $\mathbf{u}_{T-1}$ from π
    - the procedure verifies that $\mathbf{u}_{T-1} \in \mathcal{R}_\beta^m$, and $\mathbf{A}\mathbf{u}_{T-1} = \mathbf{y}$. If not, output bad.
    - recursively execute $\overline{\mathsf{V}}(\mathbf{A}, (\mathbf{x}, -\mathbf{G}\mathbf{u}_{T-1}, \beta, T-1), \pi, \tau')$
  - if $T = 2 \cdot t + 1 > 1$:
    - the procedure reads $\mathbf{u}_t$ from π
    - verify that $\mathbf{u}_t \in \mathcal{R}_\beta^m$. If not, $\mathcal{V}$ outputs bad.
    - it computes $r = \mathsf{H}(\tau')$ and checks equality by reading r from π. If not, outputs bad.
    - the procedure computes:
      - $\mathbf{x}' = \mathbf{x} + \mathbf{A} \cdot \mathbf{u}_t \cdot r \pmod q$
      - $\mathbf{y}' = \mathbf{y} \cdot r - \mathbf{G} \cdot \mathbf{u}_t \pmod q$
      - $\beta' = 2 \cdot \gamma_{\mathcal{R}} \cdot \beta$
    - recursively execute $\overline{\mathsf{V}}(\mathbf{A}, (\mathbf{x}', \mathbf{y}', \beta', t), \pi, \tau'')$

  At the end of the recursive execution, if the final output is $\mathsf{ok!}$, return valid. Otherwise, return fail.

To guarantee correct soundness, the proving algorithm P must be executed $\frac{\lambda}{\log(\lambda)}$ times implying the necessity of randomising the starting transcript because otherwise, since $(\mathsf{pp}, \mathbf{A}, \mathbf{x}, \mathbf{y}, \omega)$ is fixed, without R the algorithm P would be deterministic thus all the proof would be the same. To avoid this, the random element R is introduced in the transcript.

Maybe the approach is somehow minimal and not that efficient so here some additional thoughts!

The total proof length $|\pi|$ can be computed by summing all the communication costs presented in Section 3.3 since the transcript comprehends the instance $(\mathsf{pp}, \mathbf{A}, \mathbf{x})$ (better digest), a random string R and the prove-verifying communication.

To improve the proof's size, the transcript can start with the random string R plus the digest of the other elements, i.e. $\tau_0 = R \parallel \mathsf{H}_1(\mathsf{pp}\|\mathbf{A}\|\mathbf{x}\|\mathbf{y}\|R)$, where we introduce a different (collision-free) hash function H_1 with fixed-length binary output. This allows the compression of the starting transcript.

Only the reported in the transcript witnesses $\mathbf{u}_i$ must be readable by the verification procedure to be used in the different computational verifications. All the other values (e.g. starting transcript and random values) are computed so they can be verified in a more compact format (i.e. checking a digest). Nevertheless, how the transcript is defined can have security implications thus requiring additional care.

5 Applications

Present one or two applications of the previous protocol and/or system.

The PoSW primitive is one of many primitives in the timed cryptography domain which has some well-established application classes. The applications that utilise the PoSW primitives are presented.

Personal Note: if T implies a delay Δ, effectively there are other interesting application which not necessarily requires the verification step except for fast-forwarding between sequential evaluations. These applications are based on the idea of time-lock puzzles. For example, one can construct time-capsules, i.e. the "encrypting for the future" idea [BDD+21]) or turn-based consistent communication channel [BLL+21] (a-la blockchain).
The "issue" is that Δ time must be invested making the primitive realistically prohibitive to use in several applications. The "alternative" would be to have a trapdoor to quickly obtain $\mathbf{y}$ which goes against the PoSW concept. That is why I avoided presenting them, except for this personal note!

Lotteries and MPC Fairness. Guaranteeing randomness to the randomly sampled is hard and important. The main concern is that the random oracle is an ideal object while the hash/PRF are the closest objects that we commonly agree to look/be pseudo-random. When considering multi-party settings, it is of vital importance that any pseudo-random evaluation is verifiable to prevent a malicious actor from influencing the output in their favour.

If computing PoSW on some large T has controlled computationally cost/timing Δ (one can somehow fix T to identify a computational timing Δ), then the primitive can be used to run anything in the lottery class: a conductor sets up a lottery where people can register before a pre-decided registration deadline. When the deadline is over, the conductor must honestly sample a winner at random from the registered people which can verify that the sampling was done correctly.

This can be done by letting the conductor set up a PoSW, collect all the registration and use such information as the input $\mathbf{x}$ for the PoSW. Because of the primitive sequentiality and the full input being effectively available only at the deadline, if T is big enough to create an appropriate computational delay Δ, the final winner is unknown to everyone (conductor comprised) making the lottery fair. Plus, the conductor must provide a proof π which people can use to verify the correct winner's sampling without investing Δ time.

The class of lotteries contains many applications where the goal is to select an unpredictable candidate from a list. For example, one can use them for other type of consensus protocols, emulating the creation of a common-reference-string, sampling randomness in multi-party computations.

Expiring Password/Secrets This application wants to force a prover to expire a password/secret. Consider a shared secret, between a prover and verifier, $\mathbf{x}$ with a liveliness-budget of T. The verifier, starting from a secretly shared value $\mathbf{y}_T$ and T, requires the prover to slowly consume its budget by an amount $t_i \leq T$ until $\mathbf{x}$ is revealed. This is easy to achieve if the prover creates the challenge and never releases $\mathbf{x}$. When the verifier requests to verify and consume t_1, the prover provides the proof of correct t_1 evaluation from $\mathbf{y}_{T-t_1}$ until $\mathbf{y}_T$. The next request on t_2 provides the proof from $\mathbf{y}_{T-t_1-t_2}$ until $\mathbf{y}_{T-t_1}$ and so on. When $\sum t_i > T$, the prover should not be able to provide additional proofs forcing the prover to output $\mathbf{x}$ which expires the secret.

This idea is similar to a multi-stage commitment scheme where the commitments are chained and opened in sequence. The effective gain is that there is an additional possibility to quickly open t_i stages with a reduced cost for verifying the commitment.

A possible application would use the secrets $\mathbf{y}_i$ for $i \in [0, T]$ until the verifier requests to consume the budget. Since both parties have the secrets $\mathbf{x}$ and T, they both can compute ω and $\mathbf{y}_T$. Let's assume that the challenge $(\mathbf{A}, \mathbf{x}, T)$ is provided by the verifier to the prover and they use $\mathbf{y}_T$ as a shared secret.

At any moment, the verifier $\mathcal{V}$ might get suspicious that the prover $\mathcal{P}$ got hacked and lost control of the shared secret $\mathbf{y}_T$ and maybe some other intermediate value from the witness. For this reason, $\mathcal{V}$ requests to consume t evaluation to verify if $\mathcal{P}$ has enough partial evaluations or not. Both parties execute the verification of correct evaluation from $\mathbf{y}_{T-t}$ until $\mathbf{y}_T$. If valid, $\mathcal{V}$ is reassured that $\mathcal{P}$ has (at least) t pre-images of the one-way evaluation f.

From now onwards, they will use $\mathbf{y}_{T-t-1}$ as a shared secret, allowing the repetition of the consume-request until all T evaluations are consumed. At that point, the prover is unable to provide verifying proof forcing both to lose any shared secret between themselves. Observe that $\mathcal{V}$ can quickly verify if $\mathcal{P}$ has the same shared secret, e.g. any symmetrically encrypted fixed message with a derivate key from the shared secret would work.

As an added feature, executing the budget-consume query will fully expose the last shared secret which can be useful to execute a verification of some exchanged message, i.e. messages are exchanged and have a MAC computed from the secret key derived from the shared secret. By revealing the shared secret, anyone can verify the messages' correctness.

I hope the description is clear. I think I never encountered such an application but feels somehow useful or, at least, something that can have an interesting real application!

References

[AFK21] Thomas Attema, Serge Fehr, and Michael Klooß. Fiat-shamir transformation of multi-round interactive proofs. Cryptology ePrint Archive, Paper 2021/1377, 2021. `https://eprint.iacr.org/2021/1377`. URL: `https://eprint.iacr.org/2021/1377`.

[BDD+21] Carsten Baum, Bernardo David, Rafael Dowsley, Jesper Buus Nielsen, and Sabine Oechsner. TARDIS: A foundation of time-lock puzzles in UC. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 429–459. Springer, Heidelberg, October 2021. `doi:10.1007/978-3-030-77883-5_15`.

[BLL+21] Carlo Brunetta, Mario Larangeira, Bei Liang, Aikaterini Mitrokotsa, and Keisuke Tanaka. Turn-based communication channels. In Qiong Huang and Yu Yu, editors, *Provable and Practical Security*, pages 376–392, Cham, 2021. Springer International Publishing.

[BLM19] Carlo Brunetta, Bei Liang, and Aikaterini Mitrokotsa. Code-based zero knowledge PRF arguments. In Zhiqiang Lin, Charalampos Papamanthou, and Michalis Polychronakis, editors, *ISC 2019*, volume 11723 of *LNCS*, pages 171–189. Springer, Heidelberg, September 2019. `doi:10.1007/978-3-030-30215-3_9`.

[KS23] Abhiram Kothapalli and Srinath Setty. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573, 2023. `https://eprint.iacr.org/2023/573`. URL: `https://eprint.iacr.org/2023/573`.

[LM23] Russell W. F. Lai and Giulio Malavolta. Lattice-based timed cryptography. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part V*, volume 14085 of *LNCS*, pages 782–804. Springer, Heidelberg, August 2023. doi:10.1007/978-3-031-38554-4_25.

[LM24] Russell W. F. Lai and Giulio Malavolta. Lattice-based timed cryptography. Cryptology ePrint Archive, Paper 2024/540, 2024. URL: https://eprint.iacr.org/2024/540, doi:10.1007/978-3-031-38554-4_25.