How to Claim a Computational Feat

Clémence Chevignard[1,4], Rémi Géraud-Stewart[1,2], Antoine Houssais[1], David Naccache[1], and Edmond de Roffignac[1,3]

[1] ÉNS (DI), ISG, CNRS, PSL Research University, Paris, France.
[2] QPSI, Qualcomm Inc., San Diego CA, USA.
[3] BDS, Atos, Bezons, France
[4] CentraleSupélec, Gif-sur-Yvette, France

Abstract. Consider some user buying software or hardware from a provider. The provider claims to have subjected this product to a number of tests, ensuring that the system operates nominally. How can the user check this claim without running all the tests anew?

The problem is similar to checking a mathematical conjecture. Many authors report having checked a conjecture $C(x) = \mathsf{True}$ for all x in some large set or interval U. How can mathematicians challenge this claim without performing all the expensive computations again?

This article describes a non-interactive protocol in which the prover provides (a digest of) the computational trace resulting from processing x, for randomly chosen $x \in U$. With appropriate care, this information can be used by the verifier to determine how likely it is that the prover actually checked $C(x)$ over U.

Unlike "traditional" interactive proof and probabilistically-checkable proof systems, the protocol is not limited to restricted complexity classes, nor does it require an expensive transformation of programs being executed into circuits or ad-hoc languages. The flip side is that it is restricted to checking assertions that we dub "*refutation-precious*": expected to always hold true, and such that the benefit resulting from reporting a counterexample far outweighs the cost of computing $C(x)$ over all of U.

1 Introduction

Consider as a motivational example the sequence defined by the following iteration:

$$\phi : x \mapsto \begin{cases} x/2 & \text{if } x \equiv 0 \pmod 2 \\ 3x+1 & \text{if } x \equiv 1 \pmod 2. \end{cases}$$

starting from an initial value $x_0 \in \mathbb{N}$. The celebrated and still open *Collatz conjecture* [Col86] states that for all integers x_0, the iteration eventually reaches 1: $\forall x_0 \in \mathbb{N}, \exists n \in \mathbb{N}$ such that $\phi^n(x_0) = 1$.

An impressive body of mathematics has been produced to try and prove (or refute) this claim. Computer verification checked it to be true up to 87×2^{60} [Bař21]. Other famous mathematical conjectures verified over large intervals include the

Goldbach conjecture (verified for $x \leq 4 \times 10^{18}$, [eSHP14]) and the Riemann hypothesis (verified up to height $\simeq 3 \times 10^{12}$ [PT21]).

Such claims are however problematic from a scientific standpoint because computational exploration is, by definition, following the edge of available computing power. In theory, repeating calculations to check them just adds one bit of effort but in practice, very few researchers have access to massive infrastructures allowing to re-run such calculations. In addition, in a number of cases the verification effort is distributed, making it harder to aggregate and verify claims of computational feats.

1.1 Related work

The topic of verifiable computing was kickstarted by Babai et al. [BFLS91] in the context of monitoring large computations performed by a powerful, but faillible, supercomputer. This contrasts with the more traditional approach of majority or quorum computation, where a single task is repeated several times with the hope that not all computers conspire to lure the verifier [CRR11, CL02]. The new paradigm relies on providing a *proof of validity* together with a computational result: the celebrated PCP theorem [ALM+98, AS98, AS92, Hås01] states that with a suitably encoded proof, it is sufficient for the verifier to check three randomly chosen bits! Unfortunately, this theorem does not provide a practical, useable protocol that can be implemented. Furthermore, the PCP proof might be very long (potentially too long for the verifier to process).

An interactive protocol for verifiable computation was proposed by Ishai et al. [IKO07] and the first *non-interactive* primitives were very limited [Mic94]. A history of these developments can be found in Goldwasser et al. [GKR15]; several implementations are also available [SMBW12, VSBW13, PHGR16].

Most of the protocols above start by translating a program into a *circuit*, then translating this circuit into a polynomial (arithmetization). The verifier supplies the input and the prover executes the circuit, producing a transcript from intermediate values. Rather than sending the transcript to the verifier (who could then run the circuit themselves, and thus check the transcript's validity) the key idea is to *convince the verifier that a valid transcript exists* by encoding the transcript in some way, then having the verifier *probe* some parts of that encoded transcript. This makes it possible for a computationally weaker verifier to nevertheless check the work of a computationally stronger prover.

In the non-interactive setting literature this is achieved by either extracting a commitment [IKO07, Blu11, SBV+13, SMBW12, SVP+12, VSBW13] or by using encrypted queries [GGPR13, BCI+13, BCG+13, BCTV14], in both cases using PCP under the hood. The above thread of research shows that it is possible for the verifier to check the prover's claim — for a given program and a given input — without running the full program itself.

Our approach achieves a similar goal, albeit in a restricted setting, by different means. The core notion is that of a probabilistic counter, which is closer in spirit to the method introduced by Morris for approximate counting [Mor78, Fla85].

In the original context of approximate counting, one wishes to estimate the number of unique elements in a (large) list by using the least amount of memory; the simplest form of this algorithm consists in hashing and determining the maximum number of leading zeros after this operation [DF03, FFGM07]. While nearly optimal for the task it is set to solve [KNW10], approximate counting and its variants are easy to manipulate in an adversarial setting [PR21, RT20].

In our construction the prover will share with the verifier a small proportion of inputs $R \ll U$ on which some conjecture C holds[5], *and* which, in addition, satisfy a specific property $S(x)$. This property is designed so that the prover cannot predict in advance whether any particular value x satisfies it, until they actually compute $C(x)$ using an *agreed-upon program* P. The verifier can confirm that both $C(x)$ and $S(x)$ hold true (for instance by running P on each $x \in R$). As we will show, under some hypotheses and with appropriate parameters, the verifier then has statistical evidence that the prover did try most of the values $x \in U$.

In a sense, this mechanism combines the approximate counting approach with a refinement of the well-known proof-of-work mechanism introduced by Dwork & Naor [DN92], where a given computation $C(x)$ is being performed. However, it differs in one fundamental aspect: instead of exhibiting the *end result* of a computational task, we exhibit a number r of witnesses that the task was successful *and estimate from* r how many times the task was performed by comparing r to a threshold value $\overline{r}$.

This strategy comes with a limitation: a prover can stumble upon values x that *do not satisfy* $C(x)$. This does not contradict that the prover *actually tested* $\approx |U|$ different values of x. Therefore a prover may decide to withhold from disclosing such values x, and still get a convincing proof that they tested many value (they just so happen to "miss" those few ones). In other terms, the prover may not honestly report all the results of their computations — but they still have to be honest most of the time, otherwise verification would fail. Naturally, such counterexamples to C cannot belong to the set R transmitted to the verifier, which means that their density cannot exceed $\approx 1 - |R|/|U|$ by much. To work around this limitation, we suggest restricting the techniques discussed here to a subclass of properties that we dub "*refutation-precious*", formalized hereafter.

1.2 Incentivizing counterexample reporting

In general we have no control over the proportion of counterexamples to a generic statement C. This means that a dishonest prover could dissimulate counterexamples if they found them.

Working around this issue is possible using purely mathematical means, by randomizing and restarting the protocol many times. However in our setting this is most impractical: by design, we are handling computations that are barely feasible — we cannot hope to repeat them.

[5] This fact is denoted by $x \in R \Rightarrow C(x) = \mathsf{True}$. Typically, $U \subset \mathbb{N}$.

Alternatively, we can invoke a game-theoretical argument that provers would be better off *not withholding* counterexamples when they find them: the property to be checked is almost always true (resp. almost always false), so that an exception would be of large utility to whomever reports it. For instance, finding and reporting a counterexample to the Collatz conjecture will bring upon the finder a fame whose value is conceivably much higher than a $10M computation. In other words, no researcher would in their right mind find a Collatz counterexample and keep mum about it. We call this a *refutation-precious* claim. The same would hold for detecting a bug in a program or a circuit: it would immediately credit a researcher's reputation through the publication of a CVE.

Beware that not all situations lend themselves to such a compulsion to reveal counterexamples: for instance, an intelligence agency discovering *the very same bug as above* may be willing to stockpile exploits, and therefore keep its discovery secret.

Venturing out of cryptology into cognitive science, we can nonetheless try to nudge provers to consider counterexamples[6] $\widehat{x}$ worthy of reporting using several means:

Means	**Example**
Positive reinforcement	Prize (e.g., monetary award) if $\exists\widehat{x}$ discovered and duly reported by the tester.
Positive punishment	Penalty (e.g., a fine) if $\exists\widehat{x}$ unreported by the tester.
Negative reinforcement	Granting an immunity (e.g., from a lawsuit) if $\exists\widehat{x}$ discovered and duly reported by the tester.
Negative punishment	Revocation of a privilege (e.g., sales clearance) if $\exists\widehat{x}$ unreported by the tester.

Finally, several provers may be put in competition, making it a *prisoners' dilemma* for all of them to conspire and withhold counterexamples.

2 Preliminaries

2.1 Machine model and state

As part of our protocol we assume that the prover and the verifier agree on a concrete computational model. We only need a way to describe this model unambiguously (so that its parameters can be agreed upon) and that, when running a program P on input x, we can obtain the sequence of states τ that the machine $\mathcal{M}(P, x)$ goes through during computation.

Any such model could be used, but for the sake of compactness and applicability, we may want to use higher-level semantics.

One well-studied model that can be used is TinyRAM, introduced by Ben-Sasson et al. [BCG+13] for the very purpose of proving program execution.

[6] Formally, a counterexample is $\widehat{x} \in U$ such that $C(\widehat{x}) = \mathsf{False}$.

TinyRAM is close enough to real programs that it can be translated and compiled on most computer architectures, yet it enjoys a full specification together with a small instruction set, making it easier to prove statements about. In particular, for our needs, the TinyRAM assembly is very succinct, having only 29 opcodes, but most importantly its state is straightforward to capture.

We recall here some elementary facts about TinyRAM, for the sake of completeness[7]. A TinyRAM machine is described by two integers (W, K) together with a state $(P, \mathsf{pc}, \{r_1, \ldots, r_K\}, \mathsf{f}, \mathsf{mem}, x)$ where:

Notation	Definition
P	the program to be executed (considered as a read-only sequence of elementary operations)
pc	is a W-bit integer (indicating which instruction is currently being executed)
$r_1, \ldots, r_K$	are W-bit registers
f	a one-bit flag
mem	an array of 2^W bytes
x	a string of W-bit integers, representing the input

At every clock cycle, TinyRAM fetches the instruction in P indicated by pc, and reads if necessary from the input tape x. A special instruction answer takes a single argument and acts as the return value of program P — it immediately terminates execution. Before the execution of P all registers, all memory cells, the flag and the program counter pc are set to zero. Any other computational model could be used, but TinyRAM strikes a nice balance between usability and compactness.

2.2 Resource binding

While in the most general setting $C(x)$ should return True or False, a given program may also fail to terminate. For instance, for Collatz the intermediate values may exceed available memory, or the sequence may run longer than a reasonably set time-out value, making the verification impossible on the target machine using the specific program P — it may also loop forever. To avoid issues with such cases we impose that the execution of $P(x)$ is bounded both in terms of time and memory. We do not regard this limitation as fundamental given the goal that we seek to achieve.

Because we cannot predict in advance the amount of time and memory that P will require for processing a given x, we assume that the conjectures tested by our programs are "wrapped" in a resource-binding condition. Namely, $C_{\mathcal{M}_{B,T}}(x)$

[7] The current specifications can be found here: http://www.scipr-lab.org/doc/TinyRAM-spec-2.000.pdf.

the wrapped version of the conjecture $C(x)$, is the conjunction of:

$$C_{\mathcal{M}_{B,T}}(x) := \begin{cases} C(x) = \mathsf{True} \\ P(x) \text{ terminates before } T \text{ clock cycles} \\ P(x) \text{ uses less than } B \text{ memory cells in } \mathcal{M} \end{cases}$$

We implement the wrapping directly in $\mathcal{M}_{B,T}$: if an execution exceeds a time limit T or happens. to claim at some point more than B memory cells, then $\mathcal{M}$ returns False and halts. Such an x is not included in R.

3 A first basic protocol

We now describe a first version of our protocol. As the analysis will later show, this naive first version has limitations. It is therefore only a skeleton on which we will graft improvements, and is useful to set up some terminology.

Setup. We consider that both parties have agreed on parameters $\lambda > 0$, a set U of size u, a collision-resistant hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$, a machine setup (see Section 2.1) $\mathcal{M}_{B,T}$, and constants $\kappa < 2^\lambda$, $0 < \overline{r} < u$. Furthermore both parties have agreed on a program P with domain U and a serialization of it written $[P]$.

Prover. The prover runs Algorithm 1 and transmits information (R) to the verifier.

Algorithm 1 Prover's Algorithm

procedure GenerateProof($P, U, \kappa < 2^\lambda$)
  $R \leftarrow \emptyset$
  for all $x \in U$ **do**
    Run $\mathcal{M}_{B,T}(P,x)$, collecting τ, the sequence of machine states
    if $\mathcal{M}_{B,T}(P,x)$ finishes within the (B,T) ressource binding **then**
      if $P(x) = \mathsf{False}$ **then**
        Report a refutation precious counterexample
      else if $H([P], x, \tau) < \kappa$ **then**
        Append x to R
  return R

Verifier. The verifier runs Algorithm 2 on received data and accepts (returns True) or rejects (returns False). The threshold value $\overline{r}$ appearing in this algorithm is an integer which is determined by the verifier ahead of time based on the analysis in Section 4.

Algorithm 2 Verifier's Algorithm

```
procedure CheckProof(R ⊂ U, P, κ < 2^λ, r̄ ∈ ℕ)
    if R ⊄ U or |R| < r̄ then
        return False
    for all x ∈ R do
        Run M_{B,T}(P, x), collecting τ, the sequence of machine states during execution;
        if H([P], x, τ) ≥ κ or P(x) = False or M_{B,T}(P, x) finishes within the (B, T) ressource binding then
            return False
    return True
```

Remark 1. The property $S(x)$ discussed in the introduction is realized here as the conjunction of:

$$S(x) := \begin{cases} x \in U \\ H([P], x, \tau) < \kappa \end{cases}$$

While from an information-theoretical point of view $\mathcal{M}_{B,T}(P, x)$ is entirely determined by x and P, from a *computational* point of view there is essentially no other way to obtain $\mathcal{M}_{B,T}(P, x)$ than *actually executing* the program P on x when H is collision-resistant.

Remark 2. The requirement that P returns a Boolean value is not restrictive: without loss of generality it is always possible to turn testing P into testing a predicate $P'(x)$ defined as

$$P'(x) := (\textbf{if } P(x) = m \textbf{ then return } \mathsf{True} \textbf{ else return } \mathsf{False})$$

where m is the hard-coded value representing the evaluation of the function implemented by P at x.

Remark 3. The verifier checks that $R \subset U$. We assume that this can be tested efficiently, in time $O(|R|)$. Such is the case if U is an interval or $U = \{f(1), f(2), \dots\}$ for some function f. If U is arbitrary, the verifier may still rely on a trusted helper publishing the Merkle tree of U or digitally signing each element of U.

Remark 4. The protocol is entirely deterministic, for both the prover and the verifier. This assumes that two different runs of $P(x)$ will produce identical τs. In other words, P does not use any randomness source to produce its output. For testing algorithms where randomness is necessary (e.g., computing a DSA signature) an identical PRNG should used by the prover and the verifier.

Remark 5. The protocol is entirely parallelizable for both parties. In settings where using more than one prover should be deterred, $H([P], x, \tau)$ might be substituted by $H([P], x, \tau, R_{i-1})$ where R_i is the state of R at its previous update. This does not prevent exploring in parallel $H([P], x_j, \tau, R_{i-1})$ for several x_j values but limits the degree of parallelism that the prover might benefit of.

Remark 6. At what conditions can verification succeed or fail?

Event	**Explanation**
$x \notin R$ (failure) returns False	The prover picked an input x outside of the agreed-upon domain U. This x can be discarded (removed from R) by the verifier; *.
$H([P], x, \tau) \geq \kappa$ (failure) returns False	The prover ran the computation with a valid input, but this input shouldn't have been included in the set R. Again the verifier can discard x; *.
$P(x) =$ False (failure) returns False	The prover gave a counterexample. At this point we may either accept with a different return code, or reject the entire proof as "$P(x) =$ True for all $x \in R$" doesn't hold.
$\lvert R\rvert < \overline{r}$ (failure) returns False	The prover hasn't sent enough evidence. As for the other two failures cases denoted by $\star$, the verifier may fail "softly", requesting more evidence from the prover to continue.
$\mathcal{M}$ aborted (interruption) returns False	$\mathcal{M}$ was not powerful enough to run $P(x)$ either time-wise or memory-wise. Such x values are not included in R. We expect those events to be exceptional.
all other cases (success) returns True	This is the normal outcome expected in most runs: $P(x)$ returned True and, in addition, $H([P], x, \tau) < \kappa$. Hence x was included in R.

The practical deployment of the proposed protocol is subtle and requires taking into account several caveats, e.g.:

Remark 7. It matters that P is agreed upon ahead of time. In particular, a verifier should not accept a P provided solely by the prover, as P could be written in a way purposely causing a statistical bias: for instance, P may include the assignment instruction `var=91116;` where `var` is a variable never used in the program. The prover could to re-run the protocol with a slightly different P (having a different `var` value) until — by sheer statistical fluctuation — they obtain a false positive. (One could refer to this as P-hacking[8]). Even if for each version of P the probability η to generate an R fooling CheckProof is small, the attacker only needs *one such* R to win. The attacker's success probability is hence $1 - (1 - \eta)^n$ where n is the number of versions of P tried.

That being said, we do not address this threat, as it can be circumvented in several ways:

- Firstly, conducting such an attack requires running the protocol multiple times, which costs more than honestly following the protocol once.
- Interactivity is a simple workaround. In this case the verifier sends a random challenge c at the beginning of the protocol and $H(x)$ is replaced by $H(c|x)$.
- The prover may also commit $H(P)$ into a blockchain on January 1^{st} and later assign to c the digest of blockchain's ledger on midnight January 10^{th}.

[8] See https://en.wikipedia.org/wiki/p-hacking

- P can be published and group-signed by a large enough community of users with the group signature later being used as the randomizer c.
- P can be fed into to a verifiable delay function (VDF) with the VDF's output being used as c. This will affect honest provers once but add a penalty to dishonest provers.

Remark 8. In a number of *ad hoc* cases, P and τ can be replaced by a witness ω_x. Consider for instance a conjecture of the form $\exists \omega_x, C(\omega_x, x) = \mathsf{True}$. Typically, in the case of Collatz, ω_x can be the sequence of of digits $\phi(x), \phi^2(x), \ldots, 1$ and the inclusion criterion may be corrected to:

$$S(x) := \begin{cases} x \in U \\ H(x, \omega_x) < \kappa \end{cases}$$

A more subtle example is Goldbach's conjecture stating that *"any even integer $x \geq 2$ is the sum of two primes"*. In this case ω_x is the pair of primes whose sum gives x.[9] Here, because it may happen that multiple ω_xs can be witness to the same x a common random tape must be used to prevent a dishonest prover trying more and more witnesses[10] for the same x to compensate for skipping subsets of U.

4 An analysis of the basic protocol

We consider throughout this analysis the number of executions of P performed by the prover. This quantity is unknown, therefore we model it as an integer-valued random variable N. The crux of our protocol lies in that *we can estimate* N based on the information R provided to the verifier by the prover.

Let us introduce a few notations: $u = |U|$, and for a general symbol x we denote by $\overline{x}$ a threshold value for x. There are two essential quantities of interest in the analysis of the basic protocol:

- The probability that $N \geq u$, i.e., that the prover ran the program P on at least u different inputs: we write this probability q, and it depends on the size r of R;
- The probability that valid proof exists: we write this probability η (indeed, the protocol is deterministic); it also depends on the size r of R.

In the basic protocol, the verifier accepts if and only if $r > \overline{r}$ for a given threshold value $\overline{r}$. We provide closed form formulae for q and η, and discuss how $\overline{r}$ can be chosen.

[9] This is known to be true for *large enough* x, which means that there are only a finite – albeit immense – set of values to check. Furthermore, checking a particular (ω_x, x) can be done in polynomial time.

[10] E.g., for $x = 100$ we have: $100 = 11 + 89 = 17 + 83 = 29 + 71 = 41 + 59$ etc. Naturally, testing that several witnesses can exist is a *stronger* conjecture than Goldbach's, we just observe that this *may* happen.

4.1 Computation of q

Let q be the probability that $N \geq u$ (i.e., that the prover tried u values[11]). Since q increases with $|R|$, there exists a threshold $\overline{r}$ such that $|R| \geq \overline{r}$ is equivalent to $q \geq \overline{q}$ for some $\overline{q}$. Following standard statistical notation, $\overline{q} = 1 - \epsilon$ is the proof's *power*, and the proof is more convincing when ϵ is smaller.

Modeling H as a random function, and assuming that no counterexample was found, every value $x \in U$ is selected by the prover with equal probability $p = \kappa/2^\lambda$. The distribution of N is then exactly given by the *negative binomial distribution* of parameters p and r. This distribution models the number of successes in a sequence of independent and identically distributed Bernoulli trials before a specified number r of successes occurs. If we observe r successes when repeatedly performing a Bernoulli trial with success probability p, then the probability that there were n trials is

$$\Pr[N = n] = \binom{n-1}{r-1} p^r (1-p)^{n-r}.$$

This distribution is unimodal, has finite mode, mean and variance:

$$m = r + \frac{(1-p)(r-1)}{p}, \qquad \mu = r + \frac{(1-p)r}{p}, \qquad \sigma^2 = \frac{(1-p)r}{p^2}.$$

Since $q = \Pr[N \geq u]$, its value can be computed through the cumulative distribution function for N. Indeed, $\Pr[N \geq u] = q = 1 - \Pr[N < u]$, which can be expressed as a closed-form formula using Gauss' hypergeometric function ${}_2F_1$:

$$q(p, u, r) = (1-p)^{u-r} \binom{u-1}{r-1} {}_2F_1 \left[u - r, 1 - r; 1 + u - r; 1 - p\right].$$

While this function can efficiently be approximated numerically to several thousand digits[12], it is interesting to express it in terms of the incomplete Beta function instead, which has better numerical estimates and stability

$$B(z; a, b) = \int_0^z x^{a-1}(1-x)^{b-1}\,\mathrm{d}x = \frac{z^a}{a} {}_2F_1\left(a, 1-b; a+1; z\right).$$

In our case, it also gives a more compact expression:

$$q(p, u, r) = (u - r)\binom{u-1}{r-1} B(1 - p; u - r, r). \tag{1}$$

[11] Note a very subtle distinction: "*the probability a that the prover tried u values*" is not synonymous of "*the probability a' that the prover tried all values in U*". Indeed, if $u = 10^6$ it is easy to see that if the prover skips the testing of one specific $x_i \in U$, then a' drops to 0 by definition, whereas a does not.

[12] See for instance https://dlmf.nist.gov/15.12 or [Joh19, Section 7].

Remark 9. When u is large, $q(p, u, r)$ has a sharp transition from 0 to 1 when p is fixed and r is around pu. This can be illustrated on an example: let $(u, p) = (10^7, 10^{-3})$, consider $q(p, u, pui/100)$ for $i = 90, 91, \ldots, 104$ (Table 1 and fig. 1).

Remark 10. The problem can also be looked at from a different angle: say we have u and wish a proof with some $\overline{r}$ to be accepted. Because $q(p, u, r)$ is a smooth function of p it is easy to set r to the desired value $\overline{r}$ and solve for p numerically.

Remark 11. Note that unlike zero-knowledge protocols we do not need to aim at an extremely large q (for instance $q = 1 - 2^{-80}$) because, although non-interactive, the prover's protocol is fully deterministic. This would be different if a random tape had allowed the prover to keep trying until a satisfactory proof was found.

4.2 Computation of η

The above analysis was predicated on the notion that the prover *has a proof* to submit, i.e., that they successfully collected enough evidence. From a probabilistic standpoint there is no issue, but from an operational standpoint the prover may fail to obtain a proof because a proof respecting the imposed $\overline{r}$ simply *does not exist for a given* H.

Under constant $(u, \overline{r})$, the lower p the less likely it is that a prover obtains a valid R. The probability that a prover collects r witnesses is easy to write down:

$$\Pr[|R| \geq r] = \sum_{k=r}^{u} \binom{u}{k} p^k (1-p)^{u-k}.$$

We denote by $\eta = \Pr[|R| > \overline{r}]$ the probability that a valid proof exists.

A more practical, if approximate, expression is given by the De Moivre–Laplace theorem: for very large values of u we can estimate η as

$$\eta(p, u, \overline{r}) \approx \frac{1}{2} \operatorname{erfc}\left(\frac{\overline{r} - up}{\sqrt{2up(1-p)}}\right) \tag{2}$$

The approximation gets better as u becomes large; but importantly it *underestimates* the true value.

Remark 12. When u is large, $\eta(p, u, \overline{r})$ has a sharp transition *from* 1 *to* 0 when p is fixed and $\overline{r}$ is around pu. See Tables 1 and 2.

4.3 Contradictory goals

In an ideal world, we could aim for both $q \approx 1$ and $\eta \approx 1$, meaning that the prover is guaranteed to find a proof and that this proof is very convincing. Unfortunately, as Remarks 9 and 12 hint at, this is not possible. Instead, it seems that $q + \eta \approx 1$ which prevents having good parameters with the basic protocol.

i	90	91	92	93	94
q	1.188×10^{-24}	2.894×10^{-20}	2.374×10^{-16}	6.656×10^{-13}	6.4781×10^{-10}
η	≈ 1				

i	95	96	97	98	99
q	2.227×10^{-7}	0.000028	0.001264	0.022159	0.157324
η	≈ 1	0.999967	0.998657	0.977304	0.841466

i	100	101	102	103	104
q	0.498672	0.840260	0.976765	0.998575	0.999964
η	$\frac{1}{2}$	0.158534	0.022696	0.001343	≈ 0

Table 1. q, η for $(p, u, \overline{r}) = (10^{-3}, 10^7, 100i)$ and $90 \leq i \leq 104$. Note that $100i = \frac{i}{100} \times 10^{-3} \times 10^7$.

i	90	91	92	93	94
q	1.243×10^{-24}	3.003×10^{-20}	2.444×10^{-16}	6.807×10^{-13}	6.587×10^{-10}
η	≈ 1				

i	95	96	97	98	99
q	2.253×10^{-7}	0.000028	0.001269	0.022206	0.157431
η	≈ 1		0.998651	0.977255	0.841357

i	100	101	102	103	104
q	0.498670	0.840149	0.976715	0.998568	0.999964
η	$\frac{1}{2}$	0.158643	0.022745	0.001349	0.000032

Table 2. q, η for $(p, u, \overline{r}) = (10^{-4}, 10^8, 100i)$ and $90 \leq i \leq 104$.

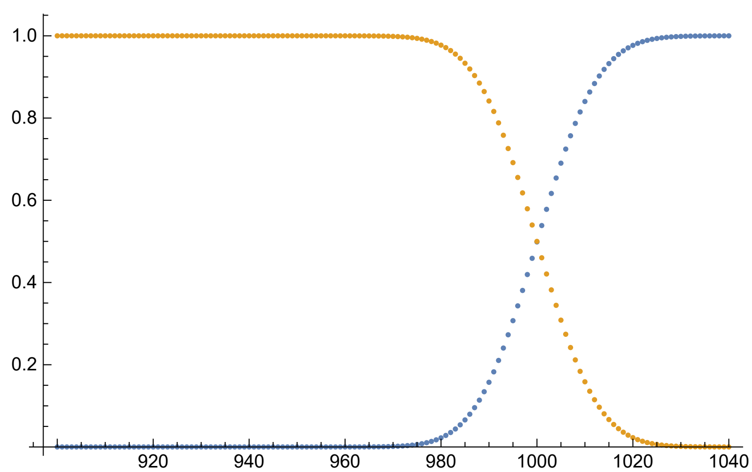

Fig. 1. $q(10^{-3}, 10^7, 10i)$ (in blue) and $\eta(10^{-3}, 10^7, 10i)$ (in orange) for $900 \leq i \leq 1040$.

To understand why, assume that we wish to impose $\eta > \overline{\eta}$. For large enough values of u there always are couples $(p, \overline{r})$ that make this true; while this problem does not admit an analytical solution, we can solve it numerically, using our approximation: $\forall \overline{\eta} \in [0,1], \exists h_0 \in \mathbb{R}$ such that

$$\frac{\overline{r} - up}{\sqrt{2up(1-p)}} < h_0 \Rightarrow \eta > \overline{\eta}.$$

For instance, for $\overline{\eta} = 0.99$ we have $h_0 = -1.64497$.

We can then express p as a function of $\overline{r}$:

$$p = \frac{h_0^2 + \overline{r} - h_0\sqrt{h_0^2 + \frac{2\overline{r}(u-\overline{r})}{u}}}{2h_0^2 + u}.$$

Decreasing p below this value will decrease η below $\overline{\eta}$. Increasing p will decrease q; therefore *this is the best value of* p *we can choose* for any given value of $\overline{r}$, under the constraint on η.

Since q is an increasing function of $\overline{r}$, and because we do not wish the verifier to test *all* inputs, the best value of q is $\overline{q}$ which is reached for $\overline{r} = u-1$. Therefore, when lower bounding η by $\overline{\eta}$ we upper bound q by $\overline{q}$ (and conversely). This gives the behavior illustrated in Figure 2 — remarkably, *this relationship does not seem to depend on the choice of* u..

We see from the above discussion that without an out-of-the-box idea, the plain strategy is essentially hopeless as the best we can get is a little improvement at the cost of having the verifier work essentially... as much as the prover.

This is a fundamental limitation of the "best effort" strategy described in Algorithms 1 and 2. In conclusion, to get out of this deadlock, we need to change the proof strategy, the verification strategy, or both.

5 Improving the basic protocol

In this section we discuss three approaches to try and overcome the limitations discussed in Section 4.3.

5.1 An intuitive idea: keep hashing

An intuitive idea consists in just keep hashing to extend R. Indeed, if a complete proof doesn't exist for a given H, maybe we can fare better with a different H?

Run the protocol twice to collect two sets R_0, R_1, where R_0 was collected with a first hash function, and R_1 with another one. Both hash functions can be agreed upon ahead of time. The prover forms $R = R_0 \cup R_1$. To see why this does not work, we can compute the probability that $N = 2u$ *given that we obtained* $|R_0| + |R_1|$ *witnesses*, in other terms, $q(p, 2u, r_0 + r_1)$. The probability of obtaining an acceptable proof in this fashion is now determined by $\eta(p, 2u, r)$: indeed, $\Pr[|R_0| + |R_1| \geq \overline{r}] = \eta(p, 2u, \overline{r})$, being the sum of two independent binomial variables of parameters (u, p).

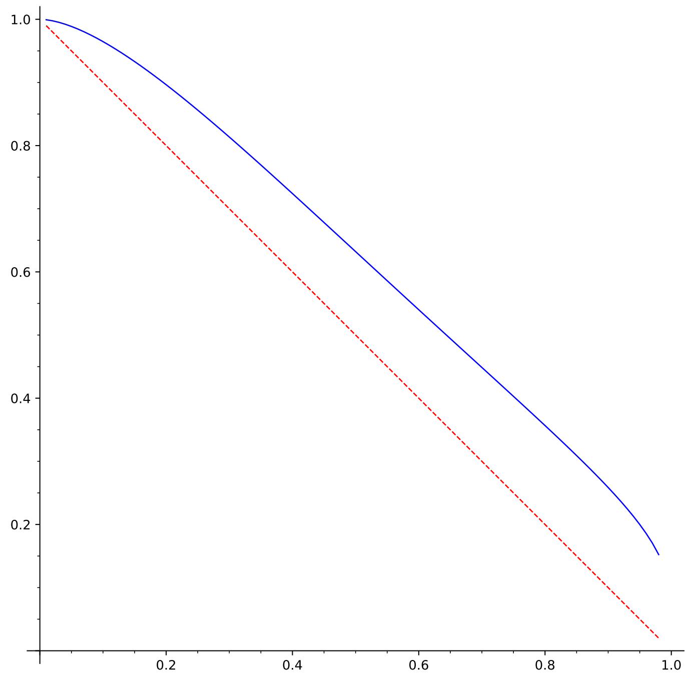

Fig. 2. Graph of $\overline{q}$ as a function of $\overline{\eta}$, for $u = 10^{10}$ (blue line). The dashed red line represents $1 - \overline{\eta}$. Plotting for $u = 10^3, \ldots, 10^{20}$ yields an identical graph.

In other terms, extending the computation by using a second hash function replaces $\eta(p, u, \overline{r})$ and $q(p, u, \overline{r})$ by respectively $\eta(p, 2u, \overline{r})$ and $q(p, 2u, \overline{r})$ *which undergo the same trade-off as the original problem*, given that this trade-off is independent of u.

It would not be better to send only the best of the two R_is, discarding the other. If the prover obtains two sets, R_0 and R_1, and sends the best of the two to the verifier we increase the prover's success probability to $1 - (1 - \eta)^2 = \eta(2 - \eta) \geq \eta$. However, simultaneously, this approach weakens the proof. The probability that a prover obtains $\overline{r}$ witnesses over a set *strictly smaller* than U is non zero. And repeating the experiment enough times this (however rare) event will eventually happen. Note that by only changing the hash function, we cannot change the value of N, therefore either both experiments were over the full input domain, or neither was. As a result, the probability that $N > \overline{r}$ drops to $q(p, u, \overline{r})^2 \leq q(p, u, \overline{r})$.

In other terms, a mere "replay" of the protocol, either by considering the union of both experiments or just taking the best, results in the same situation or in a dramatic decrease of the proof's power for a meager increase in η.

5.2 Relaxing the problem

A second strategy that can be considered is to allow some leeway in the problem statement, by recognizing as valid a proof whose domain is *slightly* different from the claim. We first discuss this idea on an example, then analyse the general case.

Experiment. Assume that we are authorized to distort reality and "extend" U in some way to a set U' of size $u' = u(1+\Delta)$ for some $\Delta > 0$. How large should such an extension be?

Set as a target a strong proof: $q(10^{-3}, 10^7, 10400) = 0.999964$. As we saw, the corresponding $\eta(10^{-3}, 10^7, 10400) \approx 0$, but this is simply because the prover falls short of having a complete proof by ≈ 500 witnesses. Allowing the prover to compute beyond 10^7, they might collect the few hundreds of missing hashes necessary to reach 10400. It appears that $\Delta = 0.07$ gives

$$\eta(10^{-3}, 1.07 \times 10^7, 10400) = 0.998144,$$

Hence, in the above example, allowing 7% more values is enough to be assured the prover can get a very convincing proof with very high probability.

Taking another numerical example: $u = 10^7$, $\eta_0 = 0.99$, $p = 10^{-4}$, $\overline{r} = 10^3$, gives $\Delta = 0.0709$, which corresponds to 709000 additional computations. In other words, we need $\Delta = 7\%$ additional *independent traces*. Note that this differs from 7% additional hashes performed on the *same traces*.

General case. More precisely, to ensure success at least η_0, the prover should claim to have checked u inputs when in reality it has checked $(1+\Delta)u$ inputs where

$$1 - \Delta \geq \frac{\alpha\left(\alpha(1-p) + \sqrt{(1-p)(\alpha^2(1-p) + 2\overline{r})}\right) + \overline{r}}{pu} \quad \text{with} \quad \alpha = \operatorname{erfc}^{-1}(2\eta_0).$$

> We thus have a first working solution: test over U but claim credit only for testing over a U' such that $|U'| = \frac{|U|}{1+\Delta}$.

The above solution might however be unfit to settings where the prover is required to test P over the *entire* set U. We hence propose a second mental experiment.

5.3 Increasing the worklard

A third strategy that can be considered is to require more work from the prover; in essence the verifier can request several independent proofs of execution on the *same* input before accepting it. Here again we first discuss an example before moving on to the general case.

Experiment. Imagine two programmers, Peter and Petra, each coding P differently as a functionally equivalent program P_b (i.e., $\forall x \in U, b \in \{0,1\}, P(x) = P_b(x)$).

For instance, if $P(x)$ is a primality test then Peter will code P as the Rabin–Miller test[13] P_0 whereas Petra will code P as the Elliptic Curve Primality Test (ECPT) P_1. It is agreed that for all practical purposes: $\forall x \in \mathbb{N}, P_0(x) = P_1(x)$.

To make sure that we are testing independent traces (i.e., independent executions), we need to assume that given a Miller–Rabin trace $\tau_0(x)$, it is impossible to infer from it an ECPT trace $\tau_1(x)$ without running the ECPT (and *vice versa*). This assumption seems very reasonable given the fundamental differences between these two primality testing approaches.

In practice, Petra will be replaced by a deterministic code obfuscation algorithm taking as input $P = P_0$ and producing P_1.

General case. The requirement on this obfuscation process is the following: An attacker should not be able to execute $P_b(x)$ and derive a trace $\tau_{\neg b}(x)$ directly from $\tau_b(x)$. Otherwise, they may skip the execution of $P_{\neg b}$. Namely, we require that any algorithm producing $\tau_b(x)$ from $\tau_{\neg b}(x)$ is more costly than running $P_{\neg b}(x)$.

To generalize this process, we proceed as described in Algorithms 3 and 4. Here $\textsc{Obfuscate}(i, P)$ is a deterministic obfuscator using the counter i to derive a pseudo-random tape[14] used to create and output P_i.

Note that a cheating prover may now test on fewer values in U and compensate the resulting drop in q by increasing i (which is the number of sessions). We hence need to bound the number of sessions below some threshold t. It remains to estimate t for some given security level $\psi = \Pr[\text{it took } t \text{ iterations to find } \gamma \text{ proofs}]$.

$$\psi = \sum_{k=\gamma}^{t} \binom{t}{k} \eta^k (1-\eta)^{t-k}.$$

Now, q is replaced by ρ:

$$\rho = 1 - \Pr\left[\forall i \in [1, \ldots, \gamma], N_i < u\right] = 1 - (1-q)^\gamma$$

Example 1. For $(p, u, \overline{r}) = (10^{-3}, 10^7, 10^4)$ we get

$$(q, \eta) = (\frac{1}{2}, 0.498672) \Rightarrow (\gamma, t) = (20, 57) \Rightarrow (\psi, \rho) = (99\%, 1 - 2^{-20})$$

We hence see that for a given u, ψ, ρ, ϵ several trade-offs between the size of the proof $O(\gamma\overline{r})$ and the work performed by the parties (resp. $O(\gamma u)$ and $O(\gamma\overline{r})$) are possible as a function of the parameters $p, \overline{r}$.

[13] With accuracy 2^{-100}.

[14] Typically by using i as a seed to a PRNG.

Algorithm 3 Prover's Algorithm

procedure GenerateParallelProof($P, U, \kappa < 2^\lambda, (\gamma, \overline{r}) \in \mathbb{N}^2$)
  $\mathbf{R} \leftarrow \emptyset$
  $\mathbf{I} \leftarrow \emptyset$
  $i \leftarrow 0$
  while $|\mathbf{R}| \neq \gamma$ **do**
    $P_i \leftarrow$ Obfuscate(i, P)
    $R_i \leftarrow$ GenerateProof(P_i, U, κ)
    if $|R_i| \geq \overline{r}$ **then**
      Append R_i to $\mathbf{R}$
      Append i to $\mathbf{I}$
    $i \leftarrow i + 1$
  return $\mathbf{R}, \mathbf{I}$

Algorithm 4 Verifier's Algorithm

procedure CheckParallelProof($\mathbf{R} \subset U^\gamma, \mathbf{I} \subset \mathbb{N}^\gamma, P, U, \kappa < 2^\lambda, (\gamma, \overline{r}, t) \in \mathbb{N}^3$)
  if $|\mathbf{R}| \neq \gamma$ or $|\mathbf{I}| \neq \gamma$ or $\max(\mathbf{I}) > t$ **then**
    return False
  for all $j \in \mathbf{I}$ **do**
    $P_j \leftarrow$ Obfuscate(j, P)
    if $\neg$CheckProof($R_j, P_j, \kappa, \overline{r}$) **then**
      return False
  return True

Remark 13. To clarify the way in which parameters are set, we refer the reader to the chronological diagram of fig. 3 where typically chosen parameters are double circled and derived parameters are single circled. An arrow from parameter x to parameter y indicates that y is derived from x. Note that in actual deployments some of the arrows might be reversed and double circles moved to other parameters. For instance the implementer may impose γ instead of p etc.

> The second solution solves the problem for the set U at the cost of slower computations for both parties.

6 Implementation

We implemented the algorithms of this paper (except those of Section 5.3) together with a virtual machine that captures the memory states of a program during execution. The code can be found at `https://github.com/Ashashin/tinyrust`.

In this particular implementation, the virtual machine takes as input a program (in the form of TinyRAM assembly source code) and a tape (in the form of a file containing data), checks the program for validity (e.g., label resolution, register indices, etc.), and runs it. For simplicity we consider $P = [P]$, i.e., the serialization *is* the source code: it is possible to use instead some canonical

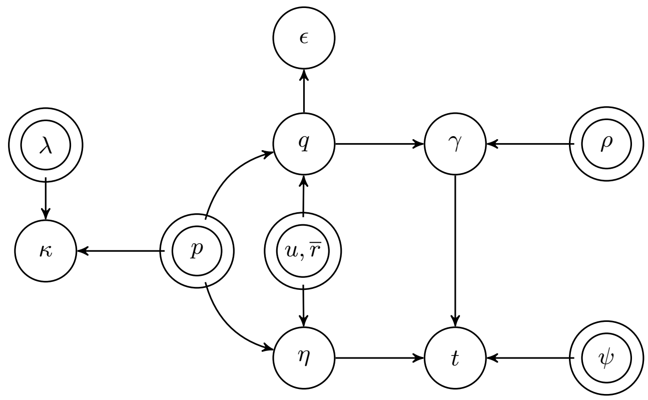

Fig. 3. Relationships between chosen (double circled) and derived (simple circled) parameters.

representation, such as an abstract syntax tree, or source code stripped of comments and non-syntactic whitespace, but committing to the byte-exact source avoids raising concerns about lexing or compiler issues.

The implementation uses for H the SHA-1 function, for which there is extensive software and hardware support, including highly optimized libraries and dedicated ASICs. This is only for demonstration purposes and a real-world implementation should use a more secure hash function.

Experimental results. To validate the analysis of Section 4 we ran a program testing the Collatz conjecture on a toy interval of size $u = 10^6$, for various parameters κ. Figure 4 plots the value of q as a function of N, that is, the *actual number of executions* performed by the prover.

7 Conclusion and further work

The technique described in this paper can find applications in a large variety of fields where counterexamples are not expected (i.e., refutation-precious). Among those are the provable testing of software against fuzzing, the verification of electronic circuits or even the testing of complex industrial control systems.

The method can also be used to support the claim that sufficiently many persons were solicited during an opinion poll: e.g. by having each person digitally sign in a deterministic way a reference string[15] and use the signature as an x.

Although not designed as such, it is also possible to apply the algorithms described in this paper to get a "useful" proof of work in blockchains. Assuming that the execution of P on an input takes work w, the average amount of work invested per proof is w/p. Note that even if w is much larger than the cost of

[15] E.g., RSA-FDH sign the string *"I, John Doe, certify that I participated in the opinion poll organized on January 19th, 2022 by the city of Grandview, Missouri."*

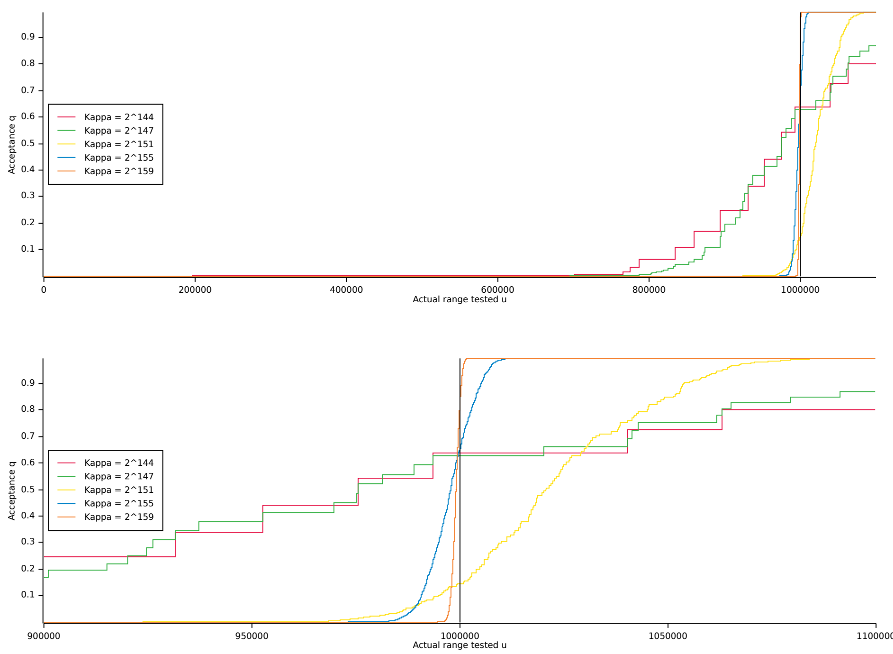

Fig. 4. q as a function of N for $u = 10^6$. Different colors correspond to different values of κ. The program in this example checks the Collatz conjecture in the interval $[1, 1.1u]$. The lower graph shows a zoomed version around $[0.9u, 1.1u]$

a unitary application of H this is not a problem as the blockchain requires to spend work anyway, no matter what the precise calculations are. In such a model, a vendor could publish a program P and its testing would reward the miners who will, in turn, be harnessed into becoming software testers. A technical detail consists in avoiding a situation where different testers test P over identical Us. This can be solved by deriving each U pseudo-randomly from the tester's identity and other current ledger parameters etc.

It may also be interesting to *simultaneously* decrease u and increase p, especially in the case that the prover ended up with too few witnesses and does not want to perform any additional effort (or there is no meaningful way to extend U). In that case, increasing p raises the number of witnesses that can be given, but always reduces the value of q (there is no "optimum" where, as p grows and more witnesses are found, q momentarily increases). This can be compensated by reducing the value u communicated to the verifier. Naturally, one wants to only minimally change the values, otherwise what the prover obtains is a very convincing proof of a very small computation. This idea calls for further analysis and fine-tuning. This and other strategies to improve the basic protocol beyond those discussed in Section 5 are likely to extend the applicability of this work.

A further extension consists in hashing not each and every memory state of $\mathcal{M}$ but skipping some states pseudo-randomly to increase efficiency by a constant factor. e.g., the parties may hash only the states at cycles $t_1, t_2, \ldots, t_i$ to get an intermediate hash h and define the next hashing milestone at time $t_{i+1} = t_i + (h \bmod 256)$. We conjecture that this does not change much in terms of security.

Finally, the verifier here does more work than necessary on each input, as it essentially runs the same program as the prover. The verifier's workload on a given input could in all likelihood be made much lighter by using verified computing techniques (cf. Section 1.1), provided that the overhead of doing so, does not nullify the advantages.

References

ALM+98. Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998. (cited on page 2)

AS92. Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs; A new characterization of NP. In *33rd Annual Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania, USA, 24-27 October 1992*, pages 2–13. IEEE Computer Society, 1992. (cited on page 2)

AS98. Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998. (cited on page 2)

Bař21. David Bařina. Convergence Verification of the Collatz problem. *The Journal of Supercomputing*, page 2681–2688, 03 2021. (cited on page 1)

BCG+13. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In Ran Canetti and Juan A. Garay, editors, *Advances in*

Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013. (cited on pages 2, 4)

BCI$^+$13. Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In Amit Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 315–333. Springer, 2013. (cited on page 2)

BCTV14. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 781–796. USENIX Association, 2014. (cited on page 2)

BFLS91. László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In Cris Koutsougeras and Jeffrey Scott Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 21–31. ACM, 1991. (cited on page 2)

Blu11. Andrew J. Blumberg. Toward practical and unconditional verification of remote computations. In Matt Welsh, editor, *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011*. USENIX Association, 2011. (cited on page 2)

CL02. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. (cited on page 2)

Col86. Lothar Collatz. On the Motivation and Origin of the $(3n+1)$-Problem. *J. of Qufu Normal University*, 12(3):9–11, 1986. (cited on page 1)

CRR11. Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 445–454. ACM, 2011. (cited on page 2)

DF03. Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*, volume 2832 of *Lecture Notes in Computer Science*, pages 605–617. Springer, 2003. (cited on page 3)

DN92. Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992. (cited on page 3)

eSHP14. Tomás Oliveira e Silva, Siegfried Herzog, and Silvio Pardi. Empirical Verification of the Even Goldbach Conjecture and Computation of Prime Gaps up to $4 \cdot 10^{18}$. *Math. Comput.*, 83(288):2033–2060, 2014. (cited on page 2)

FFGM07. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm.

In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007. (cited on page 3)

Fla85. Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, 1985. (cited on page 2)

GGPR13. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 626–645. Springer, 2013. (cited on page 2)

GKR15. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015. (cited on page 2)

Hås01. Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001. (cited on page 2)

IKO07. Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short pcps. In *22nd Annual IEEE Conference on Computational Complexity (CCC 2007), 13-16 June 2007, San Diego, California, USA*, pages 278–291. IEEE Computer Society, 2007. (cited on page 2)

Joh19. Fredrik Johansson. Computing Hypergeometric Functions Rigorously. *ACM Trans. Math. Softw.*, 45(3):30, 2019. (cited on page 10)

KNW10. Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 41–52. ACM, 2010. (cited on page 3)

Mic94. Silvio Micali. CS proofs (extended abstracts). In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 436–453. IEEE Computer Society, 1994. (cited on page 2)

Mor78. Robert H. Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978. (cited on page 2)

PHGR16. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: nearly practical verifiable computation. *Commun. ACM*, 59(2):103–112, 2016. (cited on page 2)

PR21. Kenneth G. Paterson and Mathilde Raynal. Hyperloglog: Exponentially bad in adversarial settings. *IACR Cryptol. ePrint Arch.*, page 1139, 2021. (cited on page 3)

PT21. Dave Platt and Tim Trudgian. The Riemann Hypothesis is True up to $3 \cdot 10^{12}$. *Bulletin of the London Mathematical Society*, 53(3):792–797, Jan 2021. (cited on page 2)

RT20. Pedro Reviriego and Daniel Ting. Security of hyperloglog (HLL) cardinality estimation: Vulnerabilities and protection. *IEEE Commun. Lett.*, 24(5):976–980, 2020. (cited on page 3)

SBV$^+$13. Srinath T. V. Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 71–84. ACM, 2013. (cited on page 2)

SMBW12. Srinath T. V. Setty, Richard McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. (cited on page 2)

SVP+12. Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 253–268. USENIX Association, 2012. (cited on page 2)

VSBW13. Victor Vu, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 223–237. IEEE Computer Society, 2013. (cited on page 2)